Patrones de diseño de tiempo real

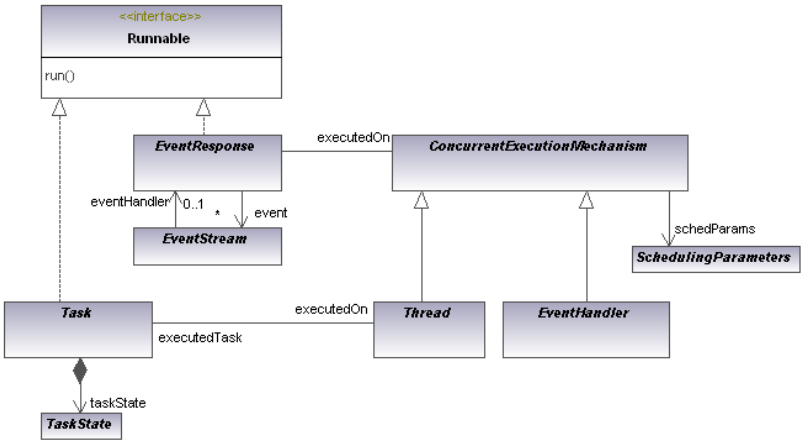
Autores: José M. Drake, Patricia López Martínez, Michael González Harbour,
Laura Barros, Cesar Cuevas y José María Martínez Lanza
Grupo de Ingeniería Software y Tiempo Real (ISTR)

Este trabajo ha sido realizado bajo la subvención del proyecto ENERGOS (Proyecto CENIT '2009)

Santander, 30 de Octubre de 2011

Glosario de términos

- **Tarea (Task)**: Actividad de duración prolongada, con estado y con intervalos de habilitación y deshabilitación (debidos a la propia naturaleza de la actividad ejecutada). Mientras está habilitada, compite con otras tareas por el acceso a recursos compartidos. Esta competición estará regulada por la política de planificación que se utiliza.
- **Stream de eventos (EventStream)**: Secuencia potencialmente infinita de eventos de un determinado tipo, que disparan la ejecución de respuestas o tareas.
- **Interrupción (Interrupt)**: Tipo especial de evento generado por una línea hardware del procesador.
- **Respuesta a evento (EventResponse)**: Actividad de vida corta, sin estado y con características de run-to-completion, esto es, una vez lanzada no se puede deshabilitar hasta su terminación. Asimismo, entre la respuesta a un evento y a otro no se guarda estado ni información.
- **Thread:** Hilo de ejecución concurrente proporcionado por el sistema operativo para la ejecución tanto de tareas como, en ocasiones, de respuestas a eventos. Los sistemas de tiempo real se diseñan como sistemas concurrentes, en los que la ejecución de los threads que ejecutan las tareas y manejadores de eventos, así como su acceso a los recursos compartidos, se controlan utilizando políticas de planificación y de sincronización dinámicas, aplicadas en tiempo de ejecución. Existen técnicas de análisis de planificabilidad que permiten determinar si en todos los escenarios de ejecución que se pueden presentar en el sistema se satisfacen todos los requisitos temporales especificados, y así mismo, existen técnicas de asignación de prioridades que permiten establecer los valores que hay que asignar a los diferentes threads y mecanismos de sincronización para que el sistema sea planificable.
- **Manejador de evento (EventHandler):** Mecanismo de ejecución concurrente proporcionado por el sistema operativo, diferente a los threads, que generalmente se utilizan para implementar respuestas a eventos hardware o temporizados.
- **Trabajo (*Job*)**: Cada una de las ejecuciones de una tarea o una respuesta a evento en un thread o manejador de evento del sistema.



Concepto y descripción de un patrón

Un patrón de diseño es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo. Son un esqueleto básico que cada diseñador adapta a las peculiaridades de su aplicación y deben cumplir las siguientes características:

- Se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.
- Deben ser reutilizables: se pueden aplicar a diferentes problemas de diseño en distintas circunstancias.

Hay diferentes puntos de vista para definir lo que es un patrón de diseño. Pueden abordar aspectos de muy bajo nivel, como ocurre en el diseño de algoritmos tales como métodos búsqueda u ordenamiento de listas. Otros plantean como patrones complejos sistemas parametrizados de determinados dominios, scadas (Supervisory Control And Data Acquisition), comunicaciones, etc. En este estudio los patrones de diseño que se tratan son descripciones de clases y objetos relacionados con la resolución de un problema de diseño general en un determinado contexto.

En general, un patrón de diseño tiene cuatro elementos esenciales:

1. El **nombre**: Permite describir mediante una o dos palabras un problemas de diseño junto con sus soluciones y consecuencias.
2. El **problema**: describe el contexto en el que aplicar el patrón
3. La **solución**: Describe los elementos que constituyen el diseño, las relaciones entre ellos, sus responsabilidades y sus colaboraciones.
4. Las **consecuencias**: Son los resultados que obtienen con su aplicación, haciendo hincapié en las ventajas e inconvenientes que se obtienen con su aplicación.

Para describir cada uno de los patrones utilizaremos las siguientes secciones:

- **Nombre**: El nombre del patrón transmite de forma sucinta su esencia. Un buen nombre es esencial ya que pasa a formar parte del vocabulario de diseño.
- **Propósito**: Una frase breve que responde a las siguientes preguntas: ¿Cuál es el problema que resuelve?, ¿Cuál es su principio de funcionamiento?.
- **Motivación**: Descripción de un escenario que ilustra el problema de diseño que resuelve. Sirve como medio de explicar su función y su modo de operar.
- **Aplicabilidad**: Describe las situaciones en que se puede aplicar el patrón de diseño

- **Estructura**: Representaciones gráficas formuladas con UML que representan los elementos que participan en el patrón, su atributos, métodos y relaciones. Puede constar de un diagrama de clases que representa su estructura estática, y diagramas de interacción que describen los principales escenarios de requerimientos, servicios y colaboraciones entre ellos.
- **Elementos**: Descripción detallada de las clases y objetos que participan en el patrón, junto con sus responsabilidades y estados.
- **Colaboraciones**: ¿Cómo consigue el patrón sus objetivos? Descripción de cómo colaboran los objetos del patrón para resolver los problemas que aborda el patrón.
- **Consecuencias**: Descripción de las ventajas, problemas y limitaciones que resultan de aplicar el patrón.
- **Implementaciones**: Especialización del patrón para su implementación sobre cada una de las plataformas de tiempo real que se consideran en este análisis: RT-POSIX y RT-Java.
- **Ejemplo y códigos**: Fragmentos de código o códigos completos de ejemplos simples en los que se utilizan el patrón.
- **Usos conocidos**: Descripción de ejemplos representativos en los que se utiliza el patrón, procurando que pertenezcan a diferentes dominios de aplicación.
- **Patrones relacionados**: Enumeración de otros patrones de de diseño que están íntimamente relacionados con el patrón que se describe, remarcando las diferencias entre ellos.

Patrones de tiempo real

En este caso se van a presentar una serie de patrones que se pueden aplicar para abordar los problemas específicos de las aplicaciones de tiempo real. Los patrones que se presentan en este documento se han dividido en cuatro grandes grupos, centrado cada uno de un aspecto característico del diseño:

I. **Patrones de control de mecanismos de ejecución concurrente**: Patrones para la creación, asignación y liberación de trabajos, control de estado y finalización de threads y manejadores.
II. **Patrones de respuesta a eventos**: Conjunto de mecanismos para la gestión de la atención y respuesta a eventos.
III. **Patrones de sincronización**: Mecanismos básicos a través de los que diferentes tareas que se ejecutan concurrentemente sincronizan sus estados y/o intercambian información de forma segura.
IV. **Patrones de tareas de tiempo real**: Describen la estructura de diferentes tareas que se ejecutan con un patrón de disparo predefinido y con unos requisitos temporales determinados**.**

En el desarrollo de un sistema de tiempo real, es habitual que se necesiten combinar patrones correspondientes a los diferentes aspectos.

I Patrones de control de mecanismos de ejecución concurrente

Conjunto de mecanismos para la creación, control de estado y finalización de threads, así como para la asignación y liberación de tareas a los mismos. Los patrones de este grupo tienen como objetivo describir diferentes mecanismos a través de los que los que se manejan los threads de un sistema de tiempo real: cómo se crean en la inicialización del sistema, se utilizan en la fase de ejecución (de tiempo real) y se eliminan en la fase de finalización.

I.1 Patrón **Objeto Activo**

I.1.1 Propósito

Ejecutar concurrentemente un conjunto de tareas que compiten por el acceso al procesador y a recursos compartidos, gestionando su prioridad para acceder a ellos. Cada objeto activo ejecuta una tarea haciendo uso de un thread propio obtenido del entorno de programación en que se implementa. El patrón formaliza la gestión de su ciclo de vida y el control de sus parámetros de planificación.

I.1.2 Motivación

Los objetos activos constituyen el elemento básico con el que se construyen las aplicaciones de tiempo real. Se utilizan para implementar y gestionar las respuestas que la aplicación proporciona a los eventos del entorno o a los eventos temporizados. Los parámetros de planificación de cada objeto activo de la aplicación deben ser configurados de forma que aquellas respuestas que son más urgentes sean ejecutadas con mayor prioridad, y con ello se reduzcan los retrasos de espera en los accesos a los recursos por los que compitan con otros objetos activos. Asimismo, resuelve problemas de seguridad en la ejecución concurrente de tareas, en los que la suspensión o finalización de la tarea sólo puede realizarse en estados de acceso a los recursos que no afecten a otras que los están requiriendo.

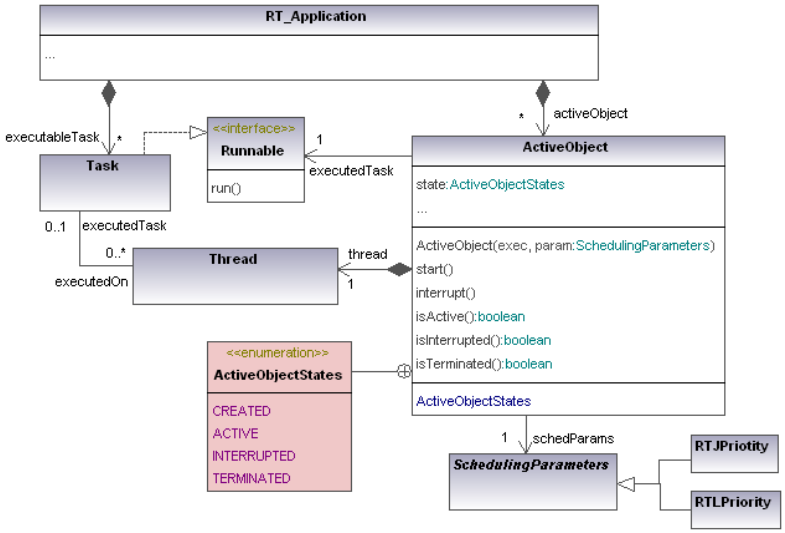
Ejemplos de uso de objetos activos:

- Una aplicación que deba muestrear una señal de forma periódica se construye mediante un objeto activo, de forma que la toma de la muestra se ejecute en el instante previsto, con independencia de otras tareas que esté ejecutando en ese instante la aplicación.
- Una aplicación que debe actualizar una base de datos con latencias especificadas debe utilizar diferentes objetos activos para ejecutar las tareas de actualización, de forma que una actualización con latencia menor no sea retrasada por otras actualizaciones que se estén ejecutando de mayor latencia.

I.1.3 Aplicabilidad

Es un patrón muy básico que se utiliza para ejecutar tareas que están definidas en el momento en el que comienza la ejecución de la aplicación.

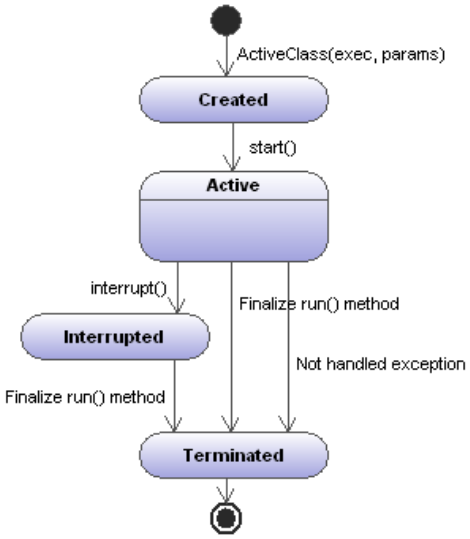
I.1.4 Estructura



I.1.5 Elementos

- *RT_Application*: Objeto que representa la aplicación que hace uso de los objetos activos. Una aplicación puede tener definidas múltiples tareas que necesita ejecutar concurrentemente, para lo cual instancia un objeto activo por cada una de ellas. Es el que controla el ciclo de vida de los objetos activos. Su ejecución no finaliza hasta que no hayan finalizado la ejecución todos los objetos activos que ha creado.

- *ActiveObject*: Clase envolvente de un thread de la plataforma de ejecución que hace homogénea su ejecución con independencia de la plataforma en la que esté instanciado. Los objetos activos son instancias de esta clase, que en su creación reciben la tarea que deben ejecutar y los parámetros con los que deben ser planificados.
  Define las siguientes asociaciones y atributos:
  - *state*: *ActiveObjectStates* => Define el estado del ciclo de vida del objeto activo. Los estados definidos son:



Estado *Created*: El objeto activo ha sido creado pero su thread interno no ha sido activado.
Estado *Active*: Se activa el thread interno, iniciandose la ejecución en él del método *run()* de la tarea asociada.
Estado *Interrupted*: Continúa la ejecución del thread interno, pero el objeto puede chequear internamente el nuevo estado para finalizar su actividad (en un punto seguro de ejecución).
Estado *Terminated*: La actividad del thread interno ha finalizado, pero el estado del objeto activo todavía puede ser testeado.

- *executedTask:Runnable* => Tarea a ejecutar por el thread.
- *thread:Thread* => Thread del sistema operativo que se encarga de la ejecución concurrente de la tarea.
- *schedParams:SchedulingParameters* => Parámetros de planificación que controlan el acceso a los recursos utilizados por la tarea asociada. Su tipo concreto debe ser compatible con el tipo de planificador que utilice la plataforma.

Define las siguientes operaciones:

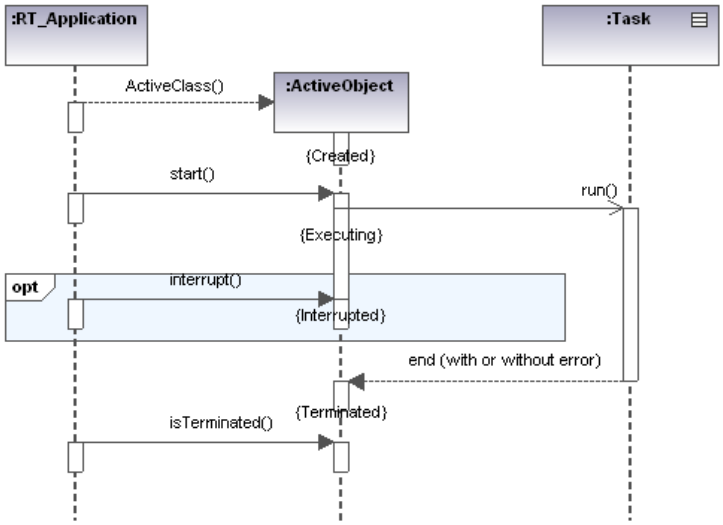
- *ActiveObject(exec:Runnable, params:SchedulingParameters)* => Constructor de un objeto activo. En la creación del objeto activo, a través de su parámetro *exec* se establece la tarea que debe ejecutar y a través de su parámetro *params* los parámetros de planificación con los que debe ser gestionado por el planificador. El objeto activo es creado en estado *Created*.
- *start():* El objeto activo pasa a estado *Active* y se inicia la ejecución concurrente del código *run()* de la tarea asociada.
- *interrupt():* El objeto activo pasa a estado *Interrupted*. El código continúa ejecutándose de acuerdo con su flujo de control, pero el estado del objeto, que es chequeado en puntos seguros de ejecución, le induce a finalizar de forma ordenada. Esta operación sustituye a otras operaciones de finalización imperativa (tipo *stop()* en Java, *abort()* en Ada o *kill()* en Linux) que son potencialmente peligrosas porque finalizan la ejecución del objeto activo sin tener en consideración los recursos que tiene tomados y pueden ser origen de diferentes tipos de bloqueos en la aplicación.
- *isInterrupted():Boolean* Retorna true si el objeto activo está en estado *Interrupted.*
- *isActive():Boolean* Retorna true si el objeto activo se encuentra en el estado *Active.*
- *isTerminated():Boolean* => Retorna true si el objeto activo está en estado *Terminated.*

- *Thread*: Thread primitivo de la plataforma de ejecución. Proporciona al objeto activo la capacidad de ejecutar concurrentemente código de la aplicación, y en ocasiones, de manejadores de eventos. Es gestionado de forma opaca por la *RT-Application.*

- *Task*: Clase que define el método y el entorno de ejecución de la actividad que ejecuta el objeto activo. Implementa la interfaz *Runnable*, que le requiere tener definido un método *run()* que define la actividad que constituye la tarea. El estado de inicialización, los manejadores de las excepciones y los resultados que genera la actividad son gestionados por el estado de la clase *Task*.
- *SchedulingParameters*: Tipo de dato abstracto que proporciona la información para la planificación del thread interno por el planificador de la plataforma de ejecución. En las plataformas que se utilizan en el proyecto las dos clases concretas que se utilizan son:
  - *RTJPriority*: Representa un entero en el rango 11..58. Es el rango de prioridades aceptadas en el nodo Java RT.
  - *RTLPriority*: Entero en el rango 1..99. Es el rango de prioridades aceptadas en el nodo RTPOSIX.

I.1.6 Colaboraciones

En el siguiente diagrama se muestra la secuencia típica de estados y acciones que conlleva el ciclo de vida de un objeto activo.



I.1.7 Consecuencias

Este patrón constituye la manera más básica de ejecutar una tarea con un comportamiento temporal predictible siempre que la carga global del sistema sea estática y conocida, y asimismo haya sido configurada para que sea planificable.

I.1.8 Modelo de tiempo real

El mapeado de este patrón a su correspondiente modelo de tiempo real es sencillo:

- Cada *ActiveObject* se modela por medio de un *Regular_Scheduling_Server*, con los parámetros de planificación *schedParams* (del tipo que corresponda según la plataforma de ejecución). Como host del *Regular_Scheduling_Server* se asigna el *Scheduler* del procesador donde se ejecute el objeto activo.
- Los *Task* ejecutados por los objetos activos se modelan como operaciones, (*Simple_Operation*, *Composite_Operation* o *Enclosing_Operation*). Para cada operación es necesario asignar los tiempos de ejecución del *Task* (en sus valores de peor, mejor y valor promedio).

El enlace entre el *Regular_Scheduling_Server* (*ActiveObject*) y la *Operation* (*Task*) se realiza en la transacción en la que se ejecute la tarea.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Elementos en el modelo del pattern objeto activo-->
<MAST_MODEL xmlns="http://mast.unican.es/xmlmast/model"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://mast.unican.es/xmlmast/model ../../Schemas/Mast_Model_A.xsd"
        Model_Name="ActiveObjectPattern" Model_Date="2010-04-01T12:00:00">

        <!-- Modelo de la plataforma de ejecución -->
        <Regular_Processor Name="theProcessor"/>
        <Primary_Scheduler Name="theScheduler" Host=" theProcessor">
                <Fixed_Priority_Scheduler Max_Priority="58" Min_Priority="11"/>
        </Primary_Scheduler>

        <!-- Modelo del thread como elemento que se puede ser planificado  -->
        <Regular_Scheduling_Server Name="theActiveObject" Scheduler=" theScheduler">
                <Fixed_Priority_Policy The_Priority="20"/>
        </Regular_Scheduling_Server>

        <!-- Modelo de la tarea como una operación simple -->
        <Simple_Operation Name="theTask"
                        Worst_Case_Execution_Time="22E-3"
                        Average_Case_Execution_Time="12.3E-3"
                        Best_Case_Execution_Time="1.42E-3"/>

        <!-- Ejemplo de la transascción que describe la actividad del objeto activo -->
        <Regular_Transaction Name="ActiveObjectActivity">
                <!-- El objeto activo ejecuta periodicamente la tarea-->
                <Periodic_External_Event Name="Trigger" Period="1.33E-3"/>
                <Regular_Event Event="End">
                        <!— Requisito temporal: deadline al final del periodo de activavión -->
                        <Hard_Global_Deadline Referenced_Event="Trigger" Deadline="1.33E-3"/>
                </Regular_Event>
                <!-- Establece la actividad que se ejecuta -->
                <Activity Input_Event="Trigger" Output_Event="EndEvent"
                        Activity_Server="theActiveObject"
                        Activity_Operation=" theTask"/>
        </Regular_Transaction>
</MAST_MODEL>
```

I.1.9 Implementaciones

- C++ con RT-POSIX:

Las tareas a ejecutar (objetos *Task*) se definen como clases que extienden a la clase abstracta *Runnable*. En esta clase se define el método *run()* sin parámetros que ejecuta el objeto activo. Cada clase activa se formulará como una clase C++, que encapsula en su interior un pthread de RT-POSIX y un objeto de tipo *Runnable*.

- RT-Java:

Las tareas a ejecutar (objetos Task) se definen en Java como clases que implementan la interfaz predefinida *Runnable*.

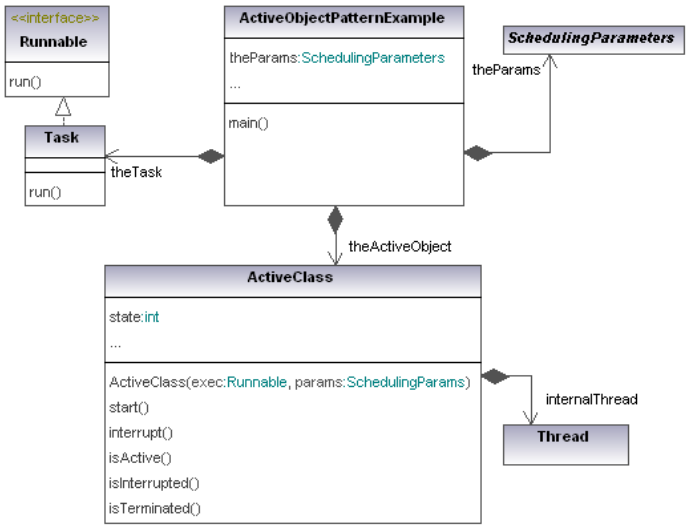
RT-Java define una clase especial para la gestión de threads de tiempo real, *RealTimeThread*, que extiende a la clase *Thread* de Java. Las características de esta clase son las siguientes:

- Implementa la interfaz *Schedulable*.
- En su constructor se le asignan los parámetros de planificación, como un objeto de la clase *SchedulingParameters*.

Cada clase activa se formulará como una clase Java normal que extiende a *RealTimeThread*, que ejecuta el objeto *Runnable*. La clase *RTJPriority* se mapea directamente a la clase *PriorityParameters* definida en RTJava.

I.1.10 Ejemplos y códigos

Para mostrar las implementaciones de este patrón en la plataforma RT-Linux y RT-Java, se utiliza un ejemplo muy simple, en el que se crea un objeto activo que durante 20 segundos ejecuta una tarea que periódicamente y con periodo de 2 s escribe en la consola "Tarea en ejecución". Su arquitectura se muestra en el siguiente diagrama de clases. Toda la aplicación se ha incluido en un único fichero para hacer el ejemplo más compacto.



- C++ con RT-POSIX:

Tabla 1: Código C++ del ejemplo de implementación del patrón ActiveObject

```
/*
 * ActiveObjectPatternExample.h
 *
 */

#ifndef ACTIVEOBJECTPATTERNEXAMPLE_H_
#define ACTIVEOBJECTPATTERNEXAMPLE_H_

#include <pthread.h>

// Clase abstracta Runnable
class Runnable
{
public:
    virtual void run() = 0;
private:
};

// Clase Task concreta que se ejecuta en el ejemplo
class Task: public Runnable {
public:
        Task(float period);
        ~Task();
        void run();
private:
        float period;
};

// Parámetros de planificación utilizados en RT-POSIX
class RTLPriority {
private:
        int priority;
public:
        RTLPriority();
        RTLPriority(int priority);
```

```
        ~RTLPriority();
        int getPriority();
};

// Estados posbiles de un objeto activo
enum ActiveObjectStates {CREATED, ACTIVE,INTERRUPTED,TERMINATED};

// Objeto activo
class ActiveClass {

public:
        Runnable *exec;
private:
        RTLPriority params;
        pthread_t internalThread;
        ActiveObjectStates state;
public:
        ActiveClass(Runnable *exec,RTLPriority params);
        ~ActiveClass();
        void start();
        void interrupt();
        bool isActive();
        void terminate();
        bool isTerminated();
};
#endif /* ACTIVEOBJECTPATTERNEXAMPLE_H_ */


/*
 * ActiveObjectPatternExample.cpp
 *
 */
#include "ActiveObjectPatternExample.h"
#include <iostream>
using namespace std;

// Métodos de la Clase Task
Task::Task(float period){
        this-> period = period;
}
Task::~Task(){}
void Task::run(){
        while (true){
                //pthread_testcancel() representa un punto seguro de ejecución.
                //Testea si el objeto ha sido suspendido, y en ese caso
                //ejecuta la terminación.
                pthread_testcancel();
                cout<<"Task en ejecucion"<<endl;
                sleep(period);
        }
}

// Métodos de la clase RTLPriority

RTLPriority::RTLPriority(){}
RTLPriority::RTLPriority(int priority){
        this->priority = priority;
}
RTLPriority::~RTLPriority(){}

int RTLPriority::getPriority(){return priority;}

//Métodos de la clase ActiveClass

//Manejador a través del que se informa al objeto activo
//de la cancelación efectiva de su thread interno.
//El objeto debe pasar a estado TERMINATED.
static void active_class_termination (void *arg)
{
        ((ActiveClass*) arg)->terminate();
}

//Método auxiliar que asocia la invocación del método run() de exec
//con un manejador (active_class_termination) que se ejecuta cuando se produce
//la cancelación efectiva del thread interno del objeto activo.
//Es el único modo de conocer cuando se realiza dicha cancelación.
static void* run(void *arg){
    pthread_cleanup_push(active_class_termination,arg);
    ((ActiveClass*) arg)->exec->run();
         pthread_cleanup_pop(1);
        return NULL;
}

ActiveClass::ActiveClass(Runnable *exec, RTLPriority params){
        this->exec = exec;
        this->params = params;
        this->state = CREATED;
}
```

```
ActiveClass::~ActiveClass(){}

void ActiveClass::start(){
        cout<<"start"<<endl;
        //Objeto de atributos para el pthread
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        int policy = SCHED_FIFO;
        struct sched_param sch_param;
        int prio =params.getPriority();
        sch_param.sched_priority = prio;
        pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED),
        pthread_attr_setschedpolicy(&attr,policy);
        pthread_attr_setschedparam(&attr,&sch_param);

        // Creación del pthread ejecutando el método auxiliar run()
        if(pthread_create(&internalThread,&attr,&run, this)==-1){
                        cout<<"Error creating thread"<<endl;
        };
        this->state = ACTIVE;
}

void ActiveClass::interrupt(){
        if (this->state == ACTIVE){
                //A través de pthread_cancel se ordena la cancelación
                //del thread interno. Ésta se producirá en algún punto
                //seguro de ejecución.
                pthread_cancel(internalThread);
                this->state = INTERRUPTED;
        }
}

bool ActiveClass::isActive(){
        return this->state == ACTIVE;
}

void ActiveClass::terminate(){
        this->state = TERMINATED;
}

bool ActiveClass::isTerminated() {
        return this->state == TERMINATED;
}

// Método principal de la aplicación, que crea el task
// y se lo pasa al objeto activo para que lo ejecute durante 20s.
// Después se cancela a través del método interrupt().
int main() {
        cout<<"Lanzo main"<<endl;
        Task* theTask = new Task(1);
        RTLPriority *theParams = new RTLPriority(8);

        ActiveClass* theActiveObject = new ActiveClass(theTask,*theParams);
        theActiveObject->start();
        sleep (20);
        theActiveObject->interrupt();
        sleep(2);
        cout<<theActiveObject->isTerminated()<<endl;
        return 0;
}
```

- RT-Java:

Tabla 2: Codigo RT-Java del ejemplo de implementación del patrón ActiveObject

```
import javax.realtime.PriorityParameters;
import javax.realtime.SchedulingParameters;
import javax.realtime.RealtimeThread;

public class ActiveObjectPatternExample {

    public static class Task implements Runnable {

        private int delay;

        public Task(int delay){
                this.delay = delay;
        }

        public void run() {
            while (!Thread.currentThread().isInterrupted()){
                System.out.println("Task en ejecuciÃ³n");
                try {
                    Thread.sleep(delay);
                } catch (InterruptedException e) {
```

```
                        System.out.println("Interrumpida dentro de sleep");
                        return;
                    }
                }
            }
        }

public static enum ActiveObjectStates{CREATED, ACTIVE, INTERRUPTED, TERMINATED}

public static class ActiveClass extends RealtimeThread {

        private Runnable exec;
        private ActiveObjectStates state;

        public ActiveClass(Runnable exec, SchedulingParameters params){
                this.exec = exec;
                this.setSchedulingParameters(params);
                this.state = ActiveObjectStates.CREATED;
        }

        public void start(){
                super.start();
                state = ActiveObjectStates.ACTIVE;
        }

        public void interrupt(){
                if (!this.isInterrupted()) {
                        super.interrupt();
                        state = ActiveObjectStates.INTERRUPTED;
                }
        }

        public boolean isActive(){
                return state == ActiveObjectStates.ACTIVE;
        }

        public boolean isInterrupted(){
                return state == ActiveObjectStates.INTERRUPTED;
        }

        public boolean isTerminated(){
                return this.getState().equals(Thread.State.TERMINATED);
        }

        public void run(){
                this.exec.run();
        }
}

public static void main(String[] args) {
        Task theTask = new Task(2000);
        SchedulingParameters theParams = new PriorityParameters(20);
        ActiveClass  theActiveObject = new ActiveClass(theTask, theParams);

        theActiveObject.start();
        try {
                Thread.sleep(20000);
        } catch (InterruptedException e) {
                }
        theActiveObject.interrupt();
        try {
                Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        System.out.println("Objeto Terminado: "+theActiveObject.isTerminated());
        System.out.println("Termina main");
}
```

I.1.11 Usos conocidos

Se puede aplicar a cualquier sistema concurrente y orientado a objetos.

I.1.12 Patrones relacionados

Este patrón representa el modo básico de implementar sistemas concurrentes, por lo que está relacionado con la práctica totalidad del resto de patrones definidos.

En el patrón únicamente se aborda la ejecución concurrente de tareas. En la mayoría de los casos dichas tareas interactuarán entre sí, requiriéndose mecanismos de sincronización para implementar la comunicación entre objetos activos. Dichos patrones se exponen en el apartado III.

Además, si se desea monitorizar el cumplimiento de requisitos temporales asociados a la tarea ejecutada se deberán utilizar algunos de los patrones de monitorización de requisitos.

I.2 Patrón **ExecutionTimeServer**

I.2.1 Propósito

Se trata de una especialización del patrón *ActiveObject*, que permite gestionar la ejecución planificable de una tarea que requiere una capacidad de procesamiento variable, en base a una capacidad de uso (*budget*) preestablecida. Si la ejecución de la tarea consume todo el budget disponible, la prioridad del thread que la ejecuta es reducida a un nivel que no afecta a otros threads con requisitos de tiempo real. El budget disponible para la ejecución se restituye de forma periódica.

I.2.2 Motivación

Existen sistemas en los que no es posible conocer la carga de trabajo completa que soporta la plataforma cuando se va a lanzar la ejecución de una aplicación que se está diseñando. Esto ocurre en plataformas abiertas, en las que no se conoce el número ni las características de las aplicaciones que se ejecutan en ellas, y en plataformas con carga de trabajo muy dinámica. En estos casos, se necesitan paradigmas de diseño de tiempo real distintos de los tradicionales, basados en un conocimiento total de la carga de la plataforma. De entre ellos, sobresale el diseño basado en reserva de recursos.

Esta estrategia es también uno de los modos más habituales de afrontar el diseño de aplicaciones donde se desea controlar el efecto que la activación de tareas aperiódicas puede tener en el resto de tareas periódicas que se ejecutan en la aplicación. Con su utilización se limita a un valor conocido el retraso que pueden sufrir las tareas periódicas debido a la atención de tareas aperiódicas.

Cada reserva es manejada por un servidor: un objeto interno del sistema operativo que actualiza el estado de la reserva en tiempo de ejecución. Un servidor es una tarea periódica cuyo propósito es servir tareas aperiódicas tan pronto como sea posible. Como cualquier tarea periódica, un servidor se caracteriza por un periodo (T) y un tiempo de computación (budget), denominado capacidad del servidor (C), disponible para sus tareas clientes (típicamente, tareas aperiódicas). Cuando un cliente se ejecuta, consume (total o parcialmente) la capacidad del servidor. El servidor tiene una política de reemplazo periódico de dicha capacidad. Si la capacidad se consume y no ha sido repuesta, el cliente no puede seguir ejecutando, por lo que esto protege a otras tareas del uso excesivo del recurso por parte de las actividades aperiódicas. En general, el servidor es planificado con el mismo algoritmo que se utiliza para las tareas periódicas, y una vez activo, sirve las tareas aperiódicas dentro del límite de su capacidad.

En función de la política de reemplazo de la capacidad existen diferentes tipos de servidores, de entre los que destacan:

- Servidor Periódico (Periodic Server):
- Servidor Diferido (Deferrable Server):
- Servidor Esporádico (Sporadic Server):

I.2.3 Aplicabilidad

- Aplicaciones con tareas aperiódicas, en las que se desea controlar la influencia que dichas tareas pueden tener en la planificación del sistema.
- Aplicaciones que se instalan en plataformas en las que no es posible conocer toda la carga de trabajo de la aplicación, con lo que tampoco se pueden aplicar las técnicas clásicas de diseño y análisis de tiempo real.

I.2.4 Estructura

I.2.5 Elementos

- Clase *ExecutionServer*: Clase abstracta que representa un servidor de ejecución utilizado en una estrategia de planificación basada en reserva de recursos. Incluye el mecanismo de la plataforma de ejecución que se va a encargar de ejecutar la tarea asignada en base a la reserva de capacidad establecida. El periodo y el budget que definen la reserva, junto a la tarea a ejecutar y los parámetros con los que debe ser planificada cuando existe capacidad disponible se reciben en la creación de cada instancia de esta clase. Cada clase concreta que lo extiende define la política que se utiliza para el reestablecimiento periódico del budget.
  Define los siguientes atributos y asociaciones, además de los heredados de *ActiveObject*:
  - *period:Time* => Periodo utilizado para el reestablecimiento de la capacidad disponible.
  - *Budget:Time* => Capacidad de ejecución máxima disponible.
  - *thread:ServerThread* => Mecanismo del sistema operativo que se encarga de la ejecución de la tarea asignada con cargo a la capacidad establecida en el servidor.

  Define las siguientes operaciones:
  - *ActiveObject(exec:Runnable, params:SchedulingParameters, period:Time, budget:Time)* => Constructor de un servidor de ejecución. A través de su parámetro *exec* se establece la tarea que debe ejecutar, a través de su parámetro *params* los parámetros de planificación con los que debe ser gestionado por el planificador, y a través de *period* y *budget* los parámetros que controlan la ejecución de la tarea en el servidor. El servidor es creado en estado *Created*.
  - *start():* Heredada de *ActiveObject*.
  - *interrupt():*Heredada de *ActiveObject.*
  - *isInterrupted():*Heredada de *ActiveObject*.
  - *isActive():Boolean* => Heredada de *ActiveObject*.
  - *isTerminated():Boolean* => Heredada de *ActiveObject*.

- *ServerThread*: Clase especializada de Thread, que representa el mecanismo primitivo de la plataforma de ejecución que proporciona al servidor la capacidad de ejecutar concurrentemente el código de la tarea al mismo tiempo que gestiona la capacidad del servidor. Es gestionado de forma opaca por la *RT-Application.*

- Clase *SporadicServer*: Clase concreta de servidor que implementa la política de gestión de la capacidad de acuerdo al modelo de servidor esporádico [].
- Clase *PeriodicServer*: Clase concreta de servidor que implementa la política de gestión de la capacidad de acuerdo al modelo de servidor periodico [].
- Clase *DeferrableServer:* Clase concreta de servidor que implementa la política de gestión de la capacidad de acuerdo al modelo de servidor diferido [].

I.2.6 Colaboraciones

I.2.7 Consecuencias

I.2.8 Modelo de tiempo real

El mapeado de este patrón a su correspondiente modelo de tiempo real es sencillo:

- Cada *ExecutionServer* se modela por medio de un *Regular_Scheduling_Server*, con los parámetros de planificación de tipo *Sporadic_Server_Policy (*tomando los valores a asignar de los schedulingParams y de los atributos budget y period del ExecutionServer). Como host del *Regular_Scheduling_Server* se asigna el *Scheduler* del procesador donde se ejecute el objeto activo.
- Los *Task* ejecutados por los servidores se modelan como operaciones, (*Simple_Operation*, *Composite_Operation* o *Enclosing_Operation*). Para cada operación es necesario asignar los tiempos de ejecución del *Task* (en sus valores de peor, mejor y valor promedio).

El enlace entre el *Regular_Scheduling_Server* (*ExecutionServer*) y la *Operation* (*Task*) se realiza en la transacción en la que se ejecute la tarea.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Elementos en el modelo del servidor de tiempo de ejecución-->
<MAST_MODEL xmlns="http://mast.unican.es/xmlmast/model"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://mast.unican.es/xmlmast/model ../../Schemas/Mast_Model_A.xsd"
        Model_Name="ExecutionTimeServerPattern" Model_Date="2011-05-23T12:00:00">

        <!-- Modelo de la plataforma de ejecución -->
        <Regular_Processor Name="theProcessor"/>
        <Primary_Scheduler Name="theScheduler" Host=" theProcessor">
                <Fixed_Priority_Scheduler Max_Priority="58" Min_Priority="11"/>
        </Primary_Scheduler>

        <!-- Modelo del thread como elemento que se puede ser planificado  -->
        <Regular_Scheduling_Server Name="theExecutionServer" Scheduler=" theScheduler">
                <Sporadic_Server_Policy Normal_Priority="20" Background_Priority="20"
                                        Initial_Capacity="15E-3" Replenishment_Period="30E-03"/>
        </Regular_Scheduling_Server>

        <!-- Modelo de la tarea como una operación simple -->
        <Simple_Operation Name="theTask"
                        Worst_Case_Execution_Time="22E-3"
                        Average_Case_Execution_Time="12.3E-3"
                        Best_Case_Execution_Time="1.42E-3"/>

        <!-- Ejemplo de la transascción que describe la actividad del objeto activo -->
        <Regular_Transaction Name="ExecutionServerActivity">
                <!-- El objeto activo ejecuta periodicamente la tarea-->
                <Sporadic_External_Event Name="Trigger" Period="30E-3"/>
                <Regular_Event Event="End">
                        <!— Requisito temporal: deadline al final del periodo de activavión -->
                        <Hard_Global_Deadline Referenced_Event="Trigger" Deadline="30E-3"/>
                </Regular_Event>
                <!-- Establece la actividad que se ejecuta -->
                <Activity Input_Event="Trigger" Output_Event="EndEvent"
                        Activity_Server="theExecutionServer"
                        Activity_Operation=" theTask"/>
        </Regular_Transaction>
</MAST_MODEL>
```

I.2.9 Implementaciones

- RT-POSIX: Con SCHED_SPORADIC y los atributos correspondientes, pero sólo se puede uno por thread. Grupos de threads no se pueden hacer, porque no es posible conocer a nivel de usuario cuándo se cambia de un thread a otro y no se podría llevar cuenta del budget que queda. Mario en MARTE le ha dado soporte a nivel de sistema operativo. (En la plataforma Energos no está implementado).
- RT-Java: Los servidores de ejecución se implementan en RT-Java a través de los *ProcessingGroupParameters*. La asociación de una instancia de esta clase a un objeto Schedulable representa la creación de un servidor. Luego, el ExecutionServer en Java se implementa como una clase que extiende a RealtimeThread, que en su interior encapsula un objeto de tipo ProcessingGroupParameters con los parámetros adecuados. Sólo está implementado en la plataforma Jamaica VM.

I.2.10 Ejemplos y códigos

Por el momento no se incluye ningún ejemplo pues no existe soporte para este tipo de patrón en ninguna de las plataformas disponibles en el proyecto.

I.2.11 Usos conocidos

Se puede aplicar a aplicaciones de tiempo real en las que concurran tareas periódicas con tareas aperiódicas. También en sistemas de tiempo real que ejecutan en plataformas con carga desconocida.

I.2.12 Patrones relacionados

I.3 Patrón **ThreadPoolExecutor**

I.3.1 Propósito

Gestionar un número limitado de mecanismos de ejecución concurrente (thread) que son generados en la fase de inicialización del sistema y a los que se puede asignar la ejecución de diferentes tareas durante la fase de ejecución de tiempo real. En la fase de finalización del sistema se gestiona su terminación y destrucción.
Las tareas que se puede asignar deben implementar la interfaz *Runnable*. Dado que se considera un paradigma orientado a objetos, la inicialización de la tarea, el retorno de los resultados y la gestión de las excepciones se realiza desde el objeto en el que está definido el método *run()*, pero este aspecto es externo al patrón. Además, el patrón proporciona medios para configurar los parámetros de planificación asociados a la ejecución de cada tarea de manera independiente.

I.3.2 Motivación

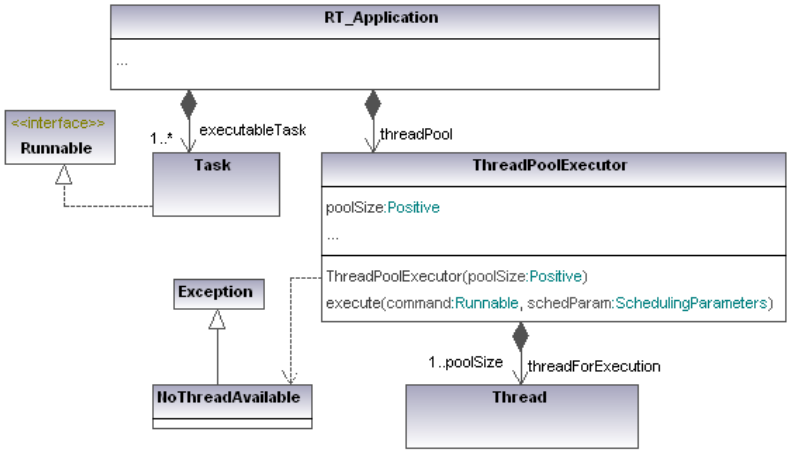
Este patrón es necesario para poder hacer planificables aquellos sistemas en los que se realizan requerimientos de ejecución concurrente de forma dinámica. Un ejemplo puede ser el gestor de una base de datos de tiempo real, en el que cada tarea que corresponde a un requerimiento recibido se debe ejecutar en un thread independiente con la prioridad que corresponda. Este sistema sólo puede hacerse planificable si todos los threads que en el peor caso puedan estar ejecutándose (igual al número máximo de concurrencia que se haya programado), son creados al inicio de la

aplicación, y al recibirse cada requerimiento sólo se asigna a uno de los threads disponibles la tarea que corresponde al requerimiento, y se inicia su ejecución con la prioridad que corresponde.

I.3.3 Aplicabilidad

Este patrón se puede aplicar a muchas de las aplicaciones de tiempo real. Todas ellas se plantean como un programa concurrente basado en un conjunto de threads, y se puede utilizar este patrón para la gestión de los mismos. Es especialmente útil su uso en los casos de aplicaciones en las que se los threads no son estáticos (se crean y asignan al inicio de la aplicación), sino que requieren una asignación dinámica de trabajos durante la fase de ejecución en las que han de establecerse requisitos de tiempo real.

I.3.4 Estructura:



I.3.5 Elementos

- *ThreadPoolExecutor*: Clase que da soporte a la ejecución asíncrona y flexible de tareas, y que independiza la definición de las tareas como unidades lógicas de trabajo, de los threads con los que las tareas se ejecutan concurrente y asíncronamente. Proporciona un número limitado (establecido a través del atributo *poolsize*) de threads para la ejecución concurrente de las invocaciones recibidas a través del método *execute*. Cuando se invoca *execute* y no hay tareas disponibles, el objeto invocante recibe la excepción *NoThreadAvailable*.

  El ciclo de vida de los threads dentro del pool es muy simple:
  - Se crean durante la inicialización de la instancia de *ThreadPoolExecutor*.
  - Se sitúan en la cola a la espera de la asignación de un trabajo.
  - Cuando se invoca *execute()*, se elige un thread de la cola de espera y se le asigna el trabajo (*command*) y los parámetros de planificación correspondientes (*schedParam*).
  - Cuando la ejecución del trabajo asignado finaliza, el thread retorna a la cola en espera de una nueva asignación.
  - Los threads son destruidos cuando se destruye el *ThreadPoolExecutor*.

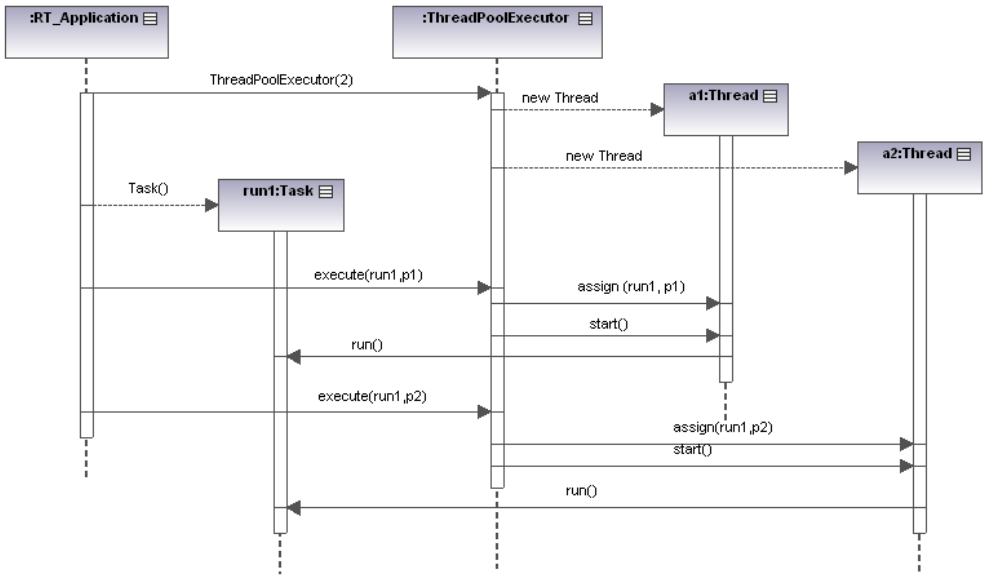
  Define los siguientes atributos y operaciones:
  - *poolsize:Positive* => Número máximo de threads que se ofrecen para la ejecución concurrente de tareas.

- *threadForExecution:Thread[1..poolSize]=>* Conjunto de threads que se encargarán de la ejecución de las tareas.
- *ThreadPoolExecutor(poolSize:Positive):* Constructor a través del que se recibe el tamaño máximo del pool.
- *execute(command:Runnable,schedParams:SchedulingParameters) raise NoAvailableThread.* Método utilizado para ordenar la ejecución de una tarea (*command*) con unos determinados parámetros de planificación (*schedParams*). Si en el momento de la invocación todos los threads del pool están ocupados y por tanto no se puede atender la petición, se eleva la excepción *NoAvailableThread.*

- *RT_Application*: Clase que representa la aplicación que hace uso del pool de threads. Una aplicación puede tener definidas múltiples tareas que necesita ejecutar concurrentemente, para lo cual instancia un objeto de tipo *ThreadpoolExecutor* cuyo tamaño deberá ser ajustado al tamaño máximo de ejecuciones concurrentes que se necesitan en la aplicación. Su ejecución no finaliza hasta que no hayan finalizado la ejecución de todos las ejecuciones concurrente que haya ordenado.

I.3.6 Colaboraciones

El siguiente diagrama de secuencia muestra como se produciría la creación de un ThreadPoolExecutor, al que posteriormente se le ordenaría la ejecución de dos tareas.



I.3.7 Consecuencias

La principal ventaja de la utilización de este patrón es que se eliminan las sobrecargas no predecibles que pueden crear la creación y destrucción dinámica de threads. Además, con él se consigue un mayor control sobre la cantidad de recursos que se utilizan en el sistema.

Además, flexibiliza el mantenimiento de programas concurrentes a lo largo de su ciclo de vida, posibilitando el cambio de las políticas de ejecución sin cambiar el código de la propia ejecución.

Al no permitirse la creación dinámica de threads, la utilización de este patrón requiere una estimación a priori del número máximo de ejecuciones concurrentes (threads) que

se necesitan para asegurar un correcto funcionamiento del sistema, tanto desde el punto de vista funcional como de tiempo real.

I.3.8 Modelo de tiempo real

El mapeado de este patrón a su correspondiente modelo de tiempo real es muy similar al del caso del objeto activo:

- Los *Task* ejecutados se modelan como operaciones, (*Simple_Operation*, *Composite_Operation* o *Enclosing_Operation*). Para cada operación es necesario asignar los tiempos de ejecución del *Task* (en sus valores de peor, mejor y valor promedio).
- Cada invocación del método *execute()* se modela a través de un *Regular_Scheduling_Server*, con los parámetros de planificación que mapean los que se pasan como argumento. Como host del *Regular_Scheduling_Server* se asigna el *Scheduler* del procesador donde se ejecute instalado el threadpool.

El enlace entre el *Regular_Scheduling_Server* (*ExecutionServer*) y la *Operation* (*Task*) se realiza en la transacción en la que se ejecute la tarea.

I.3.9 Implementaciones

- RT-POSIX:
- RT-Java: ¿Se puede usar el *ThreadPoolExecutor* en RTJava? ¿Qué pasa con las prioridades? No se cambiarle las prioridades a los threads que me da el *ThreadPoolExecutor*.

I.3.10 Ejemplos y códigos

I.3.11 Usos conocidos

I.3.12 Patrones relacionados

II Patrones de respuesta a eventos

Conjunto de mecanismos para la ejecución de respuestas a eventos que se reciben de manera asíncrona en una aplicación.

II.1 Pattern **Asynchronous Event Handler**

II.1.1 Propósito

Permite gestionar la respuesta a un evento que se produce de manera asíncrona durante la ejecución de una aplicación desde un manejador. La naturaleza de los eventos puede ser temporizada, procedente del hardware o procedente de un thread o de la red…

(Con cola/Sin cola/ Varios streams de ventos dessde el mimso manejador)

II.1.2 Aplicabilidad

Aplicaciones de tiempo real en las que se deben atender eventos que pueden ser recibidos de forma asíncrona a la ejecución.

II.1.3 Estructura

II.1.4 Elementos

II.1.5 Colaboraciones

II.1.6 Consecuencias

II.1.7 Modelo de tiempo real

II.1.8 Implementaciones
II.1.9 Ejemplos y códigos
II.1.10 Usos conocidos
II.1.11 Patrones relacionados

III Patrones de sincronización

Mecanismos básicos a través de los que diferentes tareas que se ejecutan concurrentemente en un sistema sincronizan sus estados y/o intercambian información de forma segura.

III.1 Patrón **ProtectedObject.**

III.1.1 Propósito

Conseguir un mecanismo de sincronización entre tareas que permita la ejecución de secciones críticas de código o el acceso a recursos en régimen de exclusión mutua. Garantizar que mientras una tarea accede a un recurso o actualiza una variable compartida, ninguna otra accede al mismo recurso o variable.

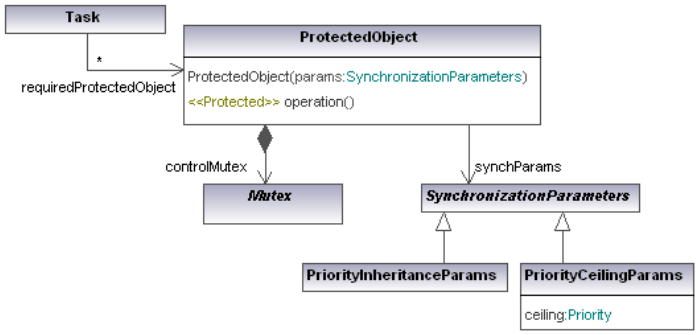
III.1.2 Motivación

Las tareas que ejecutan concurrentemente en un sistema no suelen ser independientes, sino que a lo largo de su ejecución se comunican o colaboran entre sí. Uno de los modos de comunicación que puede darse entre tareas concurrentes es el acceso a recursos compartidos, o la actualización y acceso a variables compartidas. En estos casos, es necesario utilizar algún tipo de mecanismo que asegure que dichas actualizaciones se producen de manera segura desde el punto de vista de la lógica de la aplicación, esto es, en régimen de exclusión mutua.

III.1.3 Aplicabilidad

Cualquier aplicación en la que diferentes tareas concurrentes compartan datos o recursos y se deba garantizar su coherencia y su actualización segura.

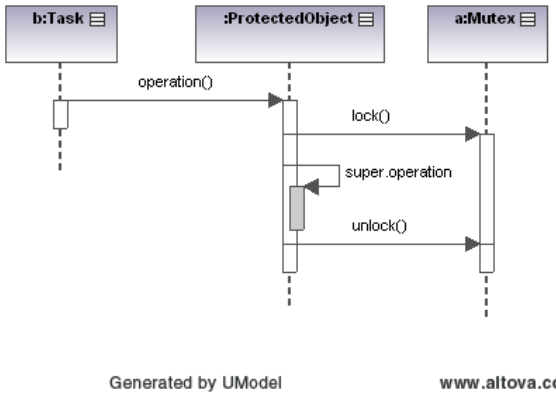
III.1.4 Estructura



III.1.5 Elementos

- *ProtectedObject*: Clase que encapsula los datos que deben ser protegidos respecto de accesos concurrentes. Estos datos sólo pueden ser accedidos a través de las operaciones del objeto, y el objeto garantiza que todas estas operaciones se realizan en régimen de exclusión mutua. Para ello define los siguientes atributos y operaciones:
  - *controlMutex: Mutex* => Objeto de tipo Mutex (proporcionado por el sistema operativo) que se utiliza para garantizar la ejecución en exclusión mutua de las

operaciones del objeto. Cuando una tarea invoca una operación en el objeto, debe tomar el mutex antes de proceder a su ejecución. Una vez terminada la ejecución de la operación, tanto en el caso normal, como si se produce un error o una excepción, el mutex debe ser liberado. Si cuando una tarea invoca una operación en el objeto protegido el mutex está tomado, la tarea se suspende a la espera de la liberación del mutex. Cuando el mutex es liberado, la tarea elegida para tomar el mutex de entre todas las que se encuentren a la espera dependerá de la política elegida. En general, será la de mayor prioridad.



El Mutex es único para todo el objeto, esto es, cuando una tarea ejecuta una operación en el objeto protegido, y en consecuencia toma el mutex, ninguna otra tarea puede ejecutar ninguna de las operaciones del objeto en concurrencia con ella.

  - *synchParams: SynchronizationParameters* => Parámetros de planificación del objeto protegido. Estos parámetros dependerán del tipo de protocolo de sincronización utilizado en la plataforma para evitar la inversión de prioridad.
  - *ProtectedObject(params: SynchronizationParameters)* => Constructor del objeto al que se le pasan los parámetros de sincronización.

- *SynchronizationParameters*: Clase abstracta que representa los parámetros de planificación que se le asocian a un objeto protegido para evitar la inversión de prioridad entre las tareas que acceden a él. En las plataformas consideradas se da soporte a dos de ellos: herencia de prioridad y techo de prioridad. Por ello se derivan dos clases concretas: *PriorityInheritance* y *PriorityCeiling*.
- *PriorityInheritance*: Parámetros de planificación de un objeto protegido que se planifica de acuerdo al protocolo de herencia de prioridad. No tiene ningún atributo asociado.
- *PriorityCeiling*: Parámetros de planificación de un objeto protegido que se planifica de acuerdo al protocolo de techo de prioridad. Tiene un único atributo *ceiling*, que representa el techo de prioridad del objeto.

III.1.6 Colaboraciones

III.1.7 Consecuencias

Utilizando este patrón se consigue la capacidad de ejecutar concurrentemente una sección crítica de código en régimen de exclusión mutua. Gracias a los protocolos de sincronización se evitan además los posibles problemas de inversión de prioridad.

III.1.8 Modelo de tiempo real

El mapeado de un *ProtectedObject* a su correspondiente modelo MAST es el siguiente:

- Se crea un *MutualExclusionResource* cuyo tipo corresponderá a la política de sincronización utilizada en la plataforma: Techo de prioridad o herencia de prioridad. Sus atributos se extraerán del atributo *params* que se asigna en la creación del objeto protegido.
- Por cada operación del objeto protegido se crea una *Operation*, cuya duración será la de la operación, y cuyo atributo *Mutex_List* será igual al objeto *MutualExclusionResource* anterior.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Elementos en el modelo de un objeto protegido-->
<MAST_MODEL xmlns="http://mast.unican.es/xmlmast/model"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://mast.unican.es/xmlmast/model ../../Schemas/Mast_Model_A.xsd"
            Model_Name="ProtectedObjectPattern" Model_Date="2011-05-23T12:00:00">

<!-- Modelo del mutex asociado al objeto protegido  -->
<Inmediate_Ceiling _Resource Name="theMutex" Ceiling="58"/>

<!-- Modelo de las operaciones del objeto operación simples -->
<Simple_Operation Name="operation" Worst_Case_Execution_Time="22E-3"
Average_Case_Execution_Time="12.3E-3" Best_Case_Execution_Time="1.42E-3">
      <Mutex_List Name="theMutex"/>
</ Simple_Operation>

</MAST_MODEL>
```

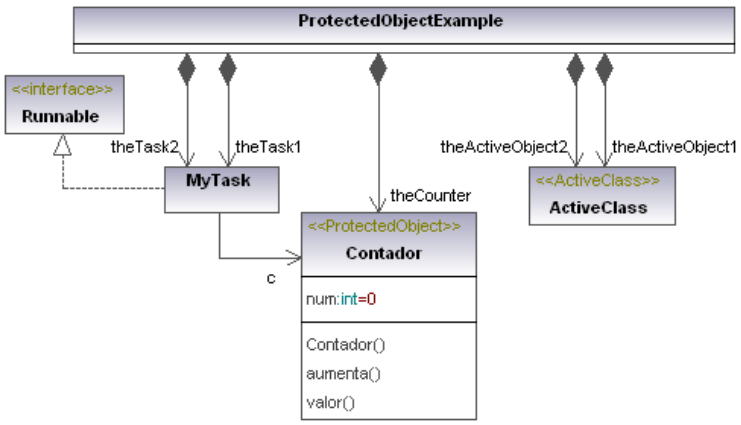
III.1.9 Implementaciones

- RT-POSIX: No existe soporte directo para objetos protegidos (como ocurre en Ada). El objeto protegido se implementará como una clase C++ que contiene un mutex POSIX (pthread_mutex_t) que se configura con los parámetros de sincronización adecuados. Todas las invocaciones de operaciones en el objeto deben tratar de tomar al mutex antes de proceder a la ejecución del código de la operación. La suspensión es implementada directamente por el mutex POSIX.
- RT-Java: Se hace uso del propio mecanismo que incluye el lenguaje, según el cual toda clase Java tiene un lock implícito, que se puede utilizar para garantizar acceso mutuamente exclusio a sus operaciones, sin más que declarar éstas como *synchronized*. El objeto protegido será por tanto una clase RT-JAVA que declarará como *synchronized* todos sus operaciones. Para asignar parámetros de sincronización se utiliza la clase *MonitorControl*, definida en la especificación RTSJ. Según la API se pueden usar las dos políticas expuestas anteriormente, sin embargo, en la JVM de la plataforma Energos sólo está disponible el protocolo de herencia de prioridad (*PriorityInheritance*).

III.1.10 Ejemplos y códigos

Para mostrar las implementaciones de este patrón en la plataforma RT-Linux y RT-Java, se utiliza un ejemplo muy simple, en el que se crea un objeto *Contador* que protege el acceso a una variable compartida entera. Cuando el objeto es inicializado la variable tiene valor 0, y a partir de ahí, diferentes tareas tienen capacidad para aumentarla en 20 unidades (de una en una). Al final se lee el valor de la variable, que

debería ser 40. Si no se utiliza el patrón, el valor que resulta no es el correcto. Como ejemplo se crean dos tareas.



III.1.11 Usos conocidos
III.1.12 Patrones relacionados

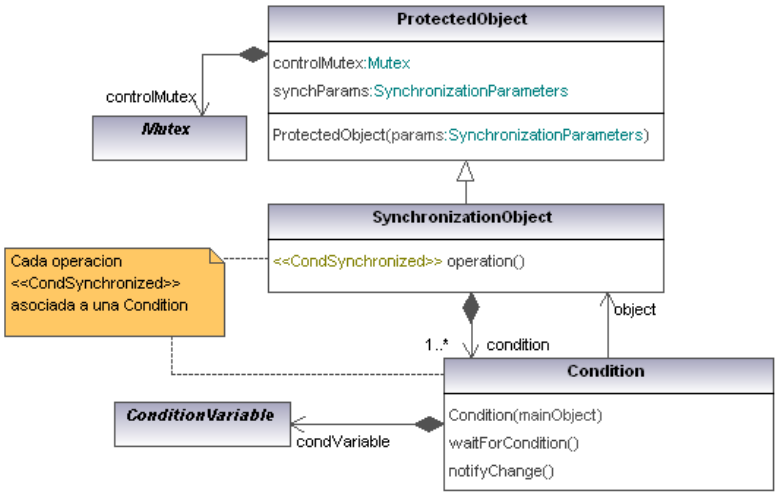
III.2 Pattern **SynchronizedObject**

III.2.1 Propósito
Conseguir un mecanismo de sincronización entre tareas que permita a una tarea quedar suspendida a la espera de que se cumpla una cierta condición o se alcance un cierto estado en un objeto. El estado del objeto varía como consecuencia de la acción de otras tareas. En el caso más simple una tarea puede quedarse suspendida a la espera de que otra tarea reactive su flujo de ejecución.

III.2.2 Motivación
Además del acceso a datos compartidos, otro de los posibles modos de interacción entre tareas concurrentes consiste en la sincronización de sus flujos de ejecución, esto es, una tarea no puede continuar ejecutando hasta que otra tarea alcanza cierto estado o hasta que se alcanza cierto estado en el sistema. La primera tarea se suspende en espera de que la segunda le comunique cuando se ha alcanzado el estado deseado. Cuando dicho estado se alcanza, la tarea suspendida reinicia su ejecución.

III.2.3 Aplicabilidad
Cualquier aplicación en la que se requiera una ejecución síncrona de tareas concurrentes.

III.2.4 Estructura

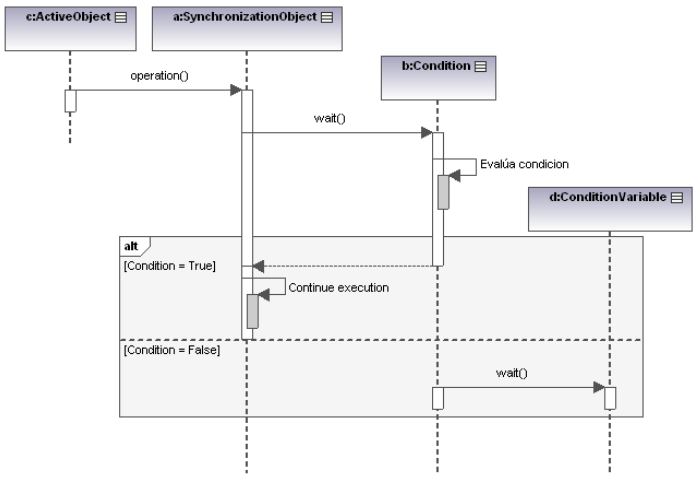


III.2.5 Elementos

- *Condition:* Clase que representa una condición en las que las tareas pueden quedarse suspendidas. Estas condiciones se expresarán en función del estado interno del objeto (de sus atributos), por lo tanto, su estado sólo puede modificarse a través de invocaciones a operaciones del objeto de sincronización. Define los siguientes atributos y operaciones:
  - *mainObject: SynchronizationObject* => Referencia al objeto de sincronización que contiene la condición, y a cuyo estado se refiere la condición. Esta referencia es necesaria para poder acceder a los atributos del objeto de sincronización.
  - *condVariable: ConditionVariable* => Mecanismo de tipo variable de condición (proporcionada por el sistema operativo) utilizado para implementar la suspensión.
  - *waitForCondition()* => Este método evalúa la condición. Si es cierta retorna inmediatamente (con lo que permite al thread invocante continuar con su ejecución). Si es falsa, suspende al thread invocante en *condVariable*.
  - *notifyChange()* => Este método evalúa la condición. Si es cierta y existen threads suspendidos en ella, se libera uno de ellos. La política para decidir qué thread se libera dependerá de la plataforma en la que nos encontremos, pero en un sistema de tiempo real deberá ser por orden de prioridad.
- *SynchronizedObject*: Objeto cuyas operaciones pueden implicar la suspensión de las tareas invocantes en función de que se cumpla cierta condición cuando son invocadas, esto es, implementan sincronización condicionada. Extiende a *ProtectedObject*, del que hereda la capacidad para que todas sus operaciones se ejecuten en régimen de exclusión mutua. Aquellas operaciones que pueden implicar suspensión condicionada se identifican por el estereotipo *<<CondSynchronized>>*. Define los siguientes atributos y operaciones:
  - *controlMutex:Mutex* => Heredado de *ProtectedObject*. Tanto la modificación como la evaluación del estado interno del objeto debe realizarse de forma segura, en exclusión mutua.
  - *condition:Condition[1..*]* => Conjunto de condiciones de guarda en las que pueden quedarse suspendidas las tareas invocantes. Cada operación del tipo *<<CondSynchronized>>* debe apuntar a una de estas condiciones. Cuando un thread invoca la operación, se invoca el método

waitForCondition() de la condición correspondiente. Si es válida, el thread prosigue con la ejecución del método, sino, se suspende en la condición a la espera de un cambio en el estado del objeto. Cuando otra tarea invoque una operación del objeto que modifica el valor de la condición, debe señalizarlo invocando el correspondiente método *notifyChange*().

III.2.6 Colaboraciones



III.2.7 Consecuencias

III.2.8 Modelo de tiempo real

El mapeado de un *SynchronizedObject* a su correspondiente modelo MAST es el siguiente:

- Se crea un *MutualExclusionResource* cuyo tipo corresponderá a la política de sincronización utilizada en la plataforma: Techo de prioridad o herencia de prioridad. Sus atributos se extraerán del atributo *params* que se asigna en la creación del objeto protegido.
- Por cada operación del objeto protegido se crea una *Operation*, cuya duración será la de la operación, y cuyo atributo *Mutex_List* será igual al objeto *MutualExclusionResource* anterior.
- *La suspensión de tareas se modelará en MAST a través del flujo de control de las transacciones. En MAST no existe ningún elemento de modelado que mapee de forma directa una suspensión de espera. El modelado deberá realizarse utilizando un elemento de tipo Join seguido de un Fork.*

III.2.9 Implementaciones

- RT-POSIX: No existe soporte directo para objetos de sincronización condicionada (como ocurre en Ada). El objeto protegido se implementará como una clase C++ que contiene un mutex POSIX (pthread_mutex_t). Cada condición se implementará como una clase C++ que engloba una variable de condición posix (pthread_cond_t).
  El acceso a las variables de condición debe realizarse siempre con el mutex asociado a la variable tomado. En este caso, el mutex que se asocia es controlMutex.

- RT-Java: Para implementar las condiciones se hace uso del propio mecanismo que incluye el lenguaje, según el cual toda clase Java tiene una variable de condición implícita, que se puede utilizar para suspender tareas en el propio objeto. Por tanto, cada condición se implementará como una clase Java, que utiliza su variable de condición implícita para suspender y señalizar tareas. El objeto de sincronización, al igual que en el caso anterior, es una clase Java que declara sus métodos como *synchronized*.

III.2.10 Ejemplos y códigos

Uno de los ejemplos más típicos de utilización de sincronización condicionada es un buffer de tamaño limitado, *BoundedBuffer*. Se trata de un mecanismo de comunicación por el que un conjunto de threads productores depositan una información para que otros threads consumidores la retiren. El mecanismo suspende a los consumidores si no hay información depositada, y a los productores si no hay espacio para depositarla. Los datos desaparecen del buffer una vez que son leídos por un consumidor. Dicho de otro modo, existen dos condiciones que deben evaluarse en el manejo del buffer:

- Un productor sólo depositará datos si el buffer no está lleno.
- Un consumidor sólo retirará datos si hay algún dato en el buffer.

III.2.11 Usos conocidos

Además del *Buffer* (o patrón Productor-Consumidor) existen otros muchos mecanismos que tienen su base en la sincronización condicionada, y que por lo tanto podrían ser construidos en base a este patrón. Ejemplos de este tipo de mecanismos son Blackboard, Signal (tanto persistente como efímera), Barrier, Broadcast, etc. Implementaciones Java de estos mecanismos se pueden consultar en [].

III.2.12 Patrones relacionados

III.3 Pattern **Socket**

III.3.1 Propósito

Conseguir un mecanismo de sincronización entre threads que se encuentran en distintos nodos. Cada *Socket* representa un extremo de un canal de comunicación entre dos nodos, que puede ser utilizado por las tareas para envíar datos o sincronizarse con tareas de nodos remotos.

III.3.2 Motivación

Los mecanismos de sincronización vistos hasta el momento se pueden utilizar únicamente entre threads que se encuentren en el mismo procesador. Sin embargo, es necesario contar con algún tipo de mecanismo que permita implementar la sincronización de espera cuando los threads que se comunican se encuentran en diferentes nodos. Con el uso de Sockets podemos conseguir que un thread en el nodo cliente se quede a la espera de que se realice algún tipo de actividad en el nodo remoto, o viceversa.

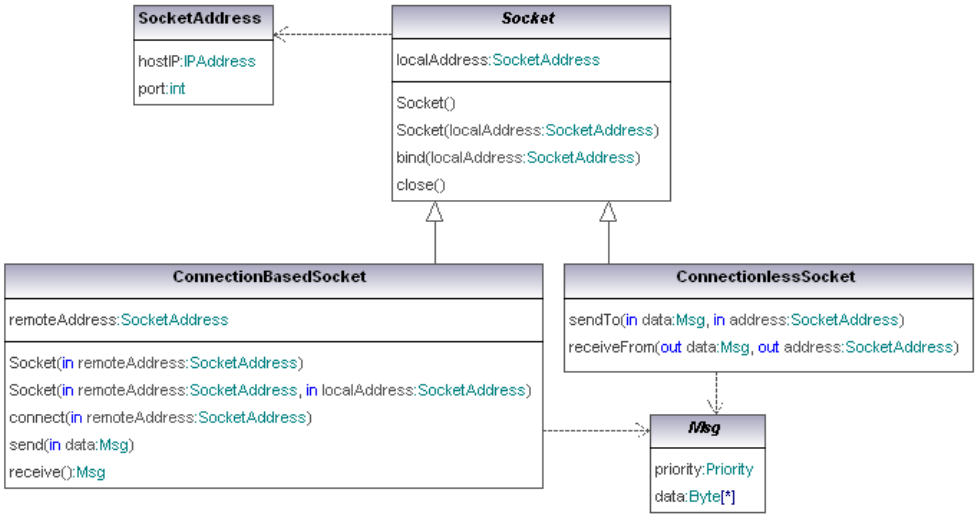
Los Sockets proporcionan una abstracción de los protocolos de transporte que se utilizan para implementar la comunicación entre los procesadores, facilitando así la

labor de los programadores, que no tienen porqué conocer detalles acerca del modo de gestionar los envíos de datos a través de la red.

III.3.3 Aplicabilidad

Cualquier aplicación en la que se requiera una ejecución síncrona de tareas que se encuentran en diferentes procesadores o procesos.

III.3.4 Estructura



III.3.5 Elementos

- *SocketAddress:* Tipo de dato que representa la dirección que se le asigna a un *Socket* y que lo identifica de forma unívoca. Resulta de la combinación de la dirección IP de la máquina en la que está instalado el *Socket* (atributo *hostID*) y de un puerto dentro de esa máquina (atributo *port*).
- *Socket:* Clase abstracta que representa cualquier tipo de socket, esto es, un terminal de comunicación entre dos máquinas. La comunicación se realiza entre pares de sockets, cada uno instalado en una máquina. La aplicación cliente realizará el envío sobre el socket instalado en su máquina, y éste se encarga de trasmitirlo al socket remoto. La ventaja de su utilización es que la aplicación no necesita conocer nada más que la propia interfaz del *Socket*.
  Define los siguientes atributos y operaciones:
  - *Socket()* => Constructor que crea el socket pero no le asigna ninguna dirección concreta.
  - *Socket (localAddress: SocketAddress)* => Constructor que crea el socket y le asigna la dirección local que se le pasa como argumento.
  - *bind(localAddress: SocketAddress)* => Asigna el socket a la dirección local que se le pasa como argumento.
  - *close()* => Cierra la conexión a través del socket.

  Un mismo *Socket* puede ser accedido concurrentemente por múltiples tareas, por lo que en su implementación debe asegurarse acceso mutuamente exclusivo a sus métodos.

  Por extensión de esta clase se da soporte a los diferentes tipos de *Socket* que existen. En este caso se definen dos clases concretas: *ConnectionBasedSocket* y

ConnectionlessSocket, que distinguen comunicación orientada a la conexión o no orientada a la conexión. El esquema de comunicación entre Sockets es diferente en ambos casos.

Sockets orientados a la conexión:

La comunicación (o intercambio de información) entre Sockets orientados a la conexión solo puede realizarse cuando el par de Sockets involucrados se han conectado previamente. Este tipo de sockets se comunica utlizando el protocolo TCP, por lo que se trata de una conexión segura: se recibirá una notificación en caso de fallo en la conexión.

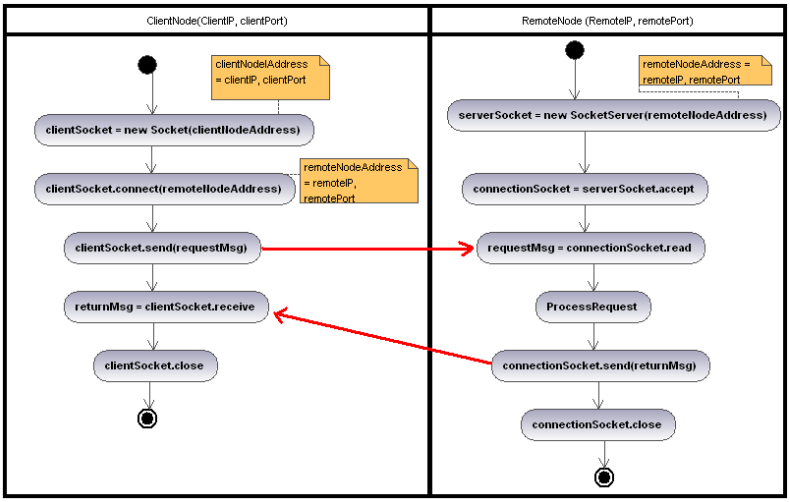
La comunicación en este caso responden a un patrón cliente servidor: hay un socket que realizan la función de servidor, esperando la recepción de conexiones por parte de los sockets clientes. Los elementos involucrados en este caso son:

- *ConnectionBasedSocket:* Clase concreta que representa un *Socket* orientado a la conexión, en el que un envío de información no puede comenzar hasta que el par de Sockets implicados en ella estén conectados. Define los siguientes atributos y operaciones:
  - *remoteAddress:SocketAddress* => Dirección del Socket remoto con el que se va a realizar la conexión.
  - *ConnectionBasedSocket(remoteAddress:SocketAddress, localAddress:SocketAddress)*=> Constructor que crea un socket asignando a la dirección local *localAddress* y conectado a la dirección remota *remoteAddress.*
  - *connect*(*remoteAddress:SocketAddress)* => Conecta el Socket a la dirección remota *remoteAddress* (al Socket asignado a dicha dirección).
  - *send(data:Msg)* => Envía el mensaje que se pasa como argumento (data) al Socket conectado a él.
  - *recv():Msg* => El Socket se queda a la espera de recibir un mensaje de su par. Se trata de una llamada bloqueante.

- *SocketServer:* Clase que representa un Socket orientado a la conexión y de tipo servidor. Una vez creado y asignado a una dirección local, queda a la espera de recibir conexiones a través del correspondiente puerto. Define los siguientes atributos y operaciones (además de los heredados de la clase raíz):
  - *accept(): ConnectionBasedSocket* => Procedimiento bloqueante, a través del cual el *SocketServer* se queda a la espera de recibir conexiones. Cuando se recibe una conexión, el procedimiento crea (y retorna) un *ConnectionBasedSocket* conectado al Socket que inició la conexión.

  El nivel de concurrencia con el que se atienden las posibles conexiones en el lado servidor es una decisión de diseño. Pueden existir diferentes esquemas u opciones:
  - Un único thread se queda a la espera de conexiones recibidas por múltiples puertos, o bien se asocia thread por SocketServer.
  - De forma ortogonal, el propio thread que recibe la conexión puede también procesarla, con lo cual no se recibirán más conexiones hasta la finalización del procesamiento. O bien, el thread únicamente se encarga de la recepción pero despierta a un thread hijo para procesarla, y se puede continuar atendiendo peticiones.

El esquema de una conexión entre Sockets de este tipo se representa en el siguiente diagrama de secuencia.

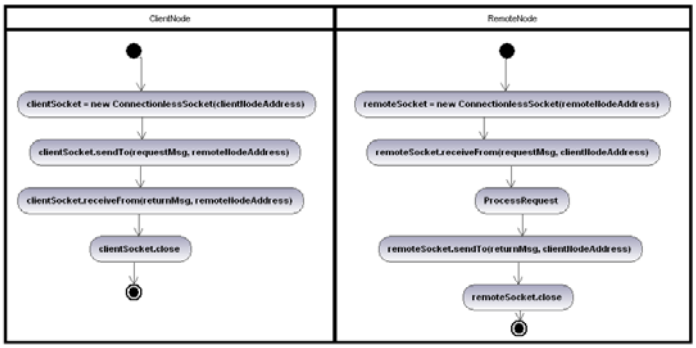


Sockets no orientados a la conexión

La comunicación (o intercambio de información) entre Sockets no orientados a la conexión no requiere la conexión previa entre los Sockets. En este caso, un mismo Socket puede realizar envíos a diferentes Sockets, especificando el Socket elegido de forma independiente para cada envío. Este tipo de sockets se comunica utilizando el protocolo UDP. Son más eficientes que los basados en TCP pero representan un mecanismo no seguro de conexión, pues no se reciben notificaciones de los posibles fallos. En este caso no se corresponde con un patrón de cliente-servidor, sino que ambos procesos son simétricos. Los elementos involucrados en este caso son:

- *ConnectionlessSocket:* Clase concreta que representa un *Socket* no orientado a la conexión. Define los siguientes atributos y operaciones:
  - *sendTo(data:Msg, address:SocketAddress)* => Envía el mensaje que se pasa como argumento (*data*) al Socket conectado a la dirección remota *address*.
  - *recvFrom(out data:Msg, out address:SocketAddress)* => El Socket se queda a la espera de recibir un mensaje (data). Cuando llega un mensaje, se recibe además la dirección del Socket que lo envió. Se trata de una llamada bloqueante.

El esquema de una conexión entre Sockets de este tipo se representa en el siguiente diagrama de secuencia.



- *Msg*: Clase que representa el mensaje que se envían a través de un Socket. En un sistema de tiempo real el envío de mensajes debe ser planificado acorde a alguna política de planificación. Los Sockets no se encargan de la planificación, será responsabilidad del sistema de comunicaciones envíar los mensajes en el orden adecuado. Para ello, el Msg que se envía a través del Socket debe incluir la prioridad (u otro parámetro de planificación) asignada a su envío.

III.3.6 Colaboraciones

III.3.7 Consecuencias

III.3.8 Modelo de tiempo real

Desde el punto de vista del modelo de tiempo real, el envío de un mensaje a través de un Socket corresponde con el envío de un mensaje a través de una red. Cada envío de un mensaje a través del Socket se modela a través de los siguientes elementos:

- Una operación de tipo *Message*, cuyos atributos es corresponderán con el tamaño de los datos a enviar.
- Un *CommunicationChannel*, cuyos parámetros de planificación corresponderán con los parámetros que se asocien al envío.

El modelo de las tareas que envían y reciben el mensaje dependerá de la lógica de la aplicación en la que se utilice el patrón (el cual se mapea al modelo de transacciones).

III.3.9 Implementaciones

RT-POSIX: La clase Socket (y todas sus clases derivadas) se implementan como un clase C++, que encapsula un Socket posix (esto es, se utiliza la librería de Sockets definida en el Standard Posix).

RTJava: Se hace uso de las clases Java *Socket* y *ServerSocket* para sockets orientados a la conexión, y la clase *DatagramSocket* para sockets no orientados a conexión.

III.3.10 Ejemplos y códigos

III.3.11 Usos conocidos

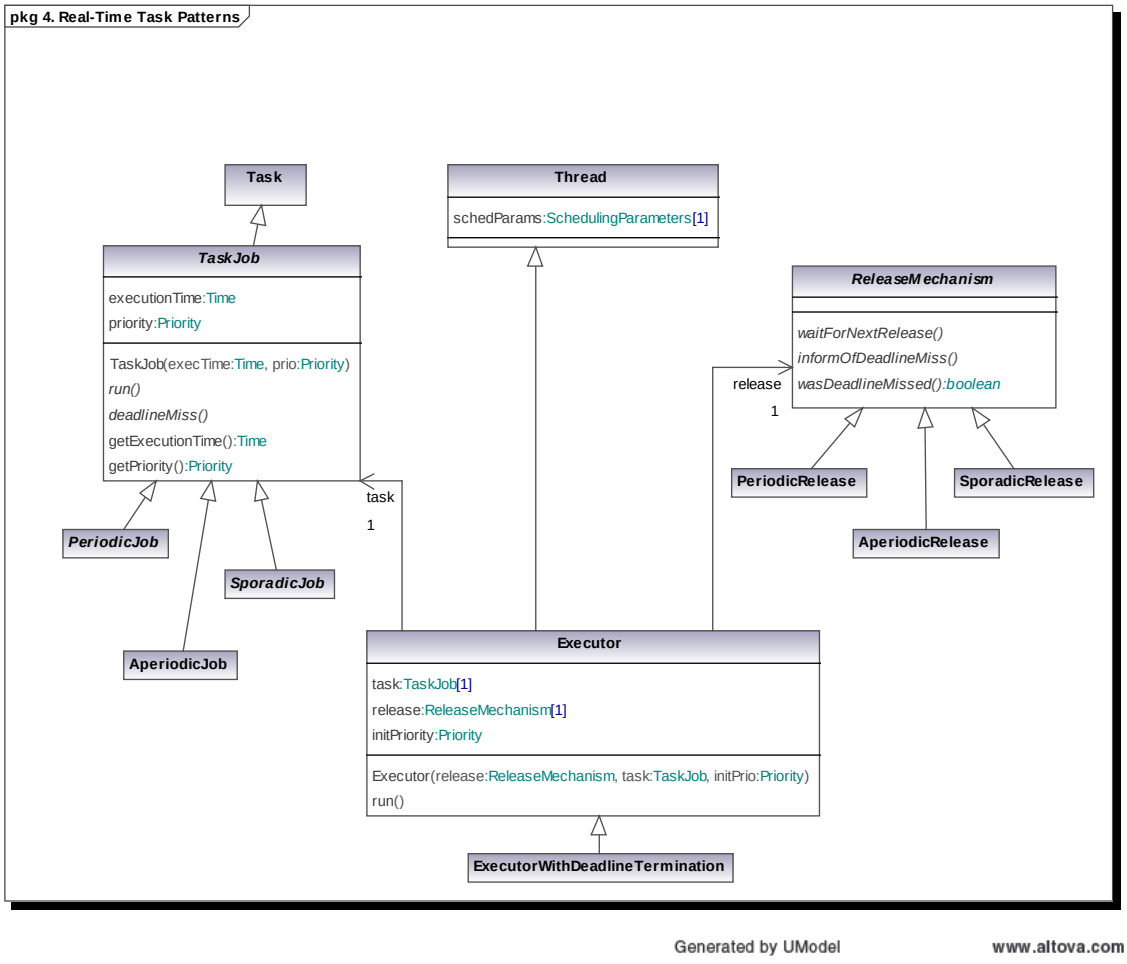
III.3.12 Patrones relacionados

IV Patrones de tareas de tiempo real

Describen la estructura de diferentes tareas que se ejecutan con un patrón de disparo predefinido y con unos requisitos temporales determinados**.** Consideramos los plazos menores o iguales que los periodos.

Elementos comunes

La figura muestra los elementos comunes a los patrones de tiempo real. Podemos ver que las clases TaskJob y ReleaseMechanism son abstractas, y se especializan según los patrones de tareas periódicas, aperiódicas y esporádicas que veremos a continuación.



Los elementos que aparecen en la figura son:

- **Executor**: Un thread que ejecuta repetitivamente un trabajo. Esta repetición puede ser periódica, aperiódica, o esporádica cuando tiene un tiempo mínimo entre activaciones. Si se detecta el incumplimiento de un plazo, se permite finalizar a la instancia que incumplió el plazo, y se invoca la operación suministrada al efecto en el TaskJob asociado.

- **ExecutorWithDeadlineTermination**: Es un Executor especializado que aborta a la instancia del TaskJob que incumple su plazo, y luego invoca la operación suministrada al efecto en el TaskJob asociado

TaskJob: Una tarea que puede ser ejecutada repetidamente por un Executor (thread de ejecución repetitiva) ante la llegada de eventos procedentes de un ReleaseMechanism. Incluye el código a ejecutar en cada instancia y el código a ejecutar si se incumple un plazo.

- **ReleaseMechanism**: Un mecanismo de activación que permite la activación repetida de un Executor para que ejecute un TaskJob. También proporciona el manejador para el caso de que se incumpla un plazo.

IV.1 Pattern **PeriodicTask**

IV.1.1 Propósito

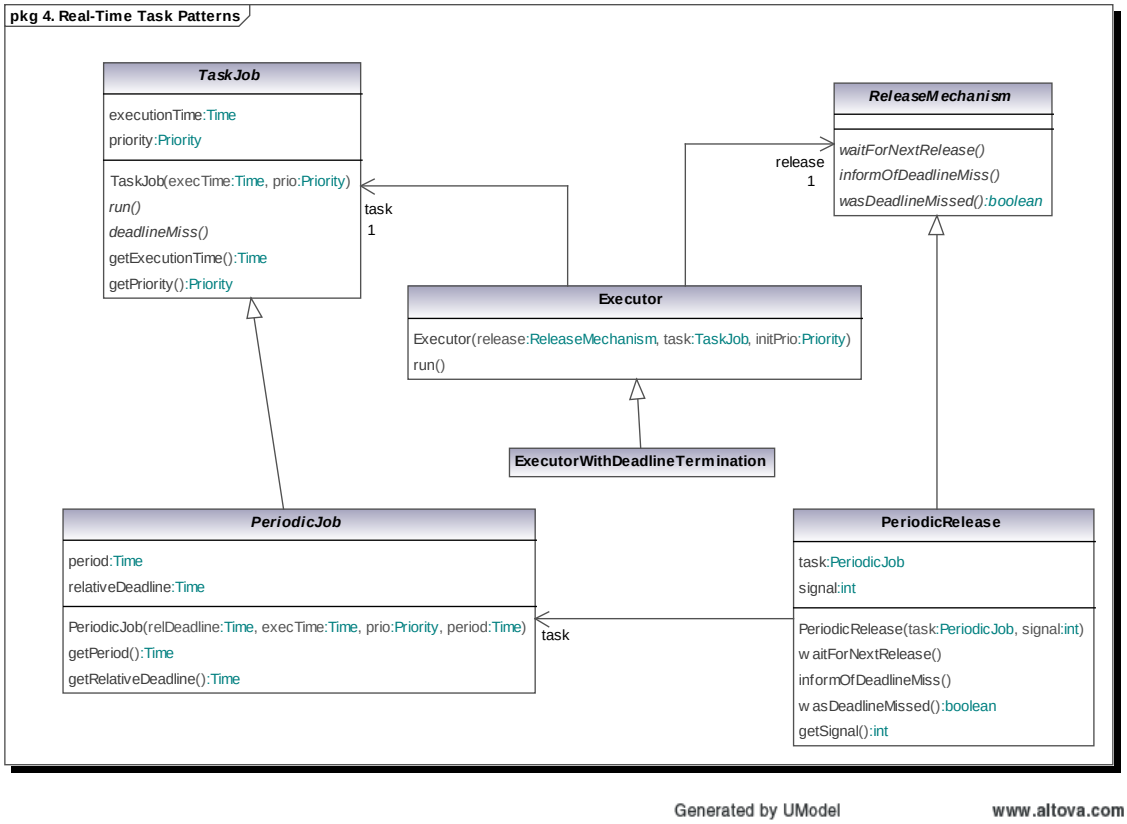
Ejecuta periódicamente un trabajo con requisitos temporales simples o múltiples. En nuestro caso consideraremos el requisito temporal simple consistente en un plazo máximo desde que se inicia el periodo hasta que finaliza la ejecución del trabajo. El patrón proporciona mecanismos para detectar incumplimientos de este plazo.

IV.1.2 Motivación. Prácticamente todas las aplicaciones de tiempo real tienen bucles de control o de monitorización que es necesario repetir de manera periódica. Es habitual que estas tareas tengan un plazo de finalización para cada una de sus activaciones, que se cuenta desde el inicio del periodo. En muchas ocasiones el plazo coincide con el final del periodo, lo que indica que cada instancia de la tarea debe finalizar antes de que comience la siguiente. En otras ocasiones el plazo es menor al periodo, lo que indica que la actividad a realizar es muy urgente, o requiere una finalización con una variabilidad (*jitter*) pequeña. También hay casos en los que el plazo puede ser mayor al periodo, aunque son menos habituales.

IV.1.3 Aplicabilidad. En subsistemas de control o de monitorización de señales.

IV.1.4 Estructura.

La figura muestra un diagrama de clases de los elementos involucrados.



IV.1.5 Elementos

a) PeriodicJob: Es una especialización del TaskJob que contiene los parámetros que caracterizan los requisitos temporales (periodo, plazo relativo y tiempo de ejecución de peor caso), los requisitos de planificación (prioridad), el código a ejecutar en cada instancia (run()) y el código a ejecutar si se detecta un incumplimiento del plazo (deadlineMiss())

   El PeriodicJob es una clase abstracta que requiere ser extendida. En la extensión deben incluirse atributos que contengan el estado de la tarea que debe preservarse de una instancia a la próxima. El constructor puede redefinirse para inicializar convenientemente este estado. Deben redefinirse los métodos abstractos run() y deadlineMiss().

b) PeriodicRelease: Es una especialización del ReleaseMechanism que produce activaciones periódicas. Su constructor requiere que se le pase un PeriodicJob y la señal a utilizar para comunicar el incumplimiento de plazo (o el mecanismo de notificación apropiado, si no se desea utilizar una señal).

   El PeriodicRelease es una clase concreta que se suministra como parte de la implementación del patrón. Sus operaciones son para uso exclusivo de la implementación, en particular del Executor asociado.

c) Executor: Proporciona el thread que ejecuta la tarea periódica. A su constructor hay que pasarle el PeriodicJob y el PeriodicRelease a utilizar.

   El Executor es una clase concreta que se suministra como parte de la implementación del patrón.

d) ExecutorWithDeadlineTermination. Es una extensión del Executor que, al igual que éste, proporciona el thread que ejecuta la tarea periódica. La principal diferencia es que establece los mecanismos necesarios para abortar la ejecución de la instancia actual del TaskJob si se detecta que ha perdido su plazo.

El ExecutorWithDeadlineTermination es una clase concreta que se suministra como parte de la implementación del patrón.

IV.1.6 Colaboraciones

La operación run() del Executor invoca constantemente a waitForNextRelease() del PeriodicRelease, para esperar al próximo periodo, y seguidamente al run() del PeriodicJob(), para ejecutar una instancia de la tarea periódica.

IV.1.7 Consecuencias

Este patrón constituye la manera más básica de ejecutar una tarea repetitiva que ejecuta una instancia periódicamente, incluyendo la posibilidad de detectar el incumplimiento del plazo de cada instancia.

IV.1.8 Implementaciones

Executor en POSIX:

```
//El constructor copia los parámetros en los atributos del objeto

void run() {
   Copiar la prioridad actual
   Establecer la prioridad a initPrio
   Crear y ejecutar un thread que ejecuta executorCode()
   Este thread se crea con política de planificación SCHED_FIFO y
   la prioridad task.getPriority()
   restablecer la prioridad original
}

executorCode() {
   while(1) {
      release.waitForNextRelease();
      task.run();
      if (release.wasDeadlineMissed()) {
         task.deadlineMissed();
      }
   }
}
```

ExecutorWithDeadlineTermination En POSIX:

```
//El constructor copia los parámetros en los atributos del objeto

void run() {
   Copiar la prioridad actual
   Establecer la prioridad a initPrio
   crea un manejador de señal para la señal release.getSignal()
   que ejecuta manejador()
   establecer la máscara de señales apropiada para el manejador
   crear y ejecutar un thread que ejecuta executorCode()
   Este thread se crea con política de planificación SCHED_FIFO y
   la prioridad task.getPriority()
   restablecer la prioridad original
}

manejador() {
   longjmp(contexto,1); // interrumpir al thread
}

executorCode() {
   while(1) {
      if (setjmp(contexto)==0) {
         // no ha habido incumplimiento de plazo
         release.waitForNextRelease(); // esperar
         // pedir al rm que mande una señal si se incumple el plazo
         release.informOfDeadlineMiss();
         task.run();
      } else {
         // hubo un incumplimiento de plazo
         task.deadlineMissed();
      }
   }
}
```

PeriodicRelease en POSIX

```
//Atributos adicionales
timer_t timer
boolean primeraVez
time period
time deadline
boolean deadlineWasMissed

PeriodicRelease(PeriodicJob task, int signal) {
   this.task=task;
   this.signal=signal;
   this.primeraVez=true
   this.period=task.getPeriod();
   this.deadline=task.getRelativedeadline();
   this.deadlineWasMissed=false;
   initialize timer based on CLOCK_MONOTONIC, with an action set to send
      this.signal
}

void waitForNextRelease() {
   if (primeraVez) {
      primeraVez=false;
      get_time(CLOCK_MONOTONIC, nextPeriod);
   } else {
       disarm_timer;
       get_time(CLOCK_MONOTONIC, now);
       deadlineWasMissed=now-nextPeriod>relativeDeadline;
       nextPeriod=nextPeriod+period;
       clock_nanosleep(CLOCK_MONOTONIC,ABS_TIME, nextPeriod);
   }
}

boolean wasDeadlineMissed() {
     return deadlineWasMissed;
}

void informOfDeadlineMiss() {
   timer_arm(timer,nextPeriod+deadline);
}

int getSignal() {
   return this.signal;
}
```

IV.1.9 Forma de uso

A continuación se muestran los pasos necesarios para utilizar este patrón.

a) Extender la clase PeriodicJob:
   a. Añadir los atributos que contendrán el estado de la tarea periódica que debe preservarse de una instancia a la siguiente.
   b. Se puede redefinir el constructor añadiéndole los parámetros que se necesiten para inicializar el estado de la tarea periódica. El constructor

debe invocar al constructor del PeriodicJob para inicializar los parámetros básicos, y luego ejecutar las instrucciones de inicialización que se deseen.

c. Redefinir la operación run() que es el código que se ejecutará a cada instancia
d. Redefinir la operación deadlineMiss(), que será invocada si hay un incumplimiento de plazo

b) Crear un objeto de la nueva clase extensión de PeriodicJob. Lo llamaremos tarea. Al crearlo, pasarle los parámetros temporales (prioridad, plazo, tiempo de ejecución), los parámetros de planificación (prioridad) y cualquier otro definido en la extensión.
c) Crear un objeto de la clase PeriodicRelease, pasándole como parámetro al constructor el objeto tarea y el número de señal a utilizar para la notificación del incumplimiento de plazo. Llamaremos al nuevo objeto release.
d) Crear un objeto de la clase Executor o ExecutorWithDeadlineTermination (según el comportamiento deseado), pasándole como parámetros los objetos tarea y release.
e) Invocar el método run() del Executor para arrancar la terea periódica, que funcionará a partir de ahora de forma indefinida.

IV.1.10 Ejemplo

En este ejemplo describimos una aplicación Java con dos tareas periódicas que, a modo de ejemplo, comparten el mismo código y ponen mensajes personalizados en pantalla de forma periódica.

```
// Extensión de PeriodicJob que contiene el estado de la tarea periódica
// y el código a ejecutar en cada instancia

public class MiTareaPeriodica extends PeriodicJob {

   // atributo que cuenta el número de instancias ejecutadas
   private int contador=0;
   // atributo que contiene el identificador de la tarea
   private int id;

   // constructor al que se le pasan el plazo, tiempo de ejecución, prioridad y
   // periodo del Job

   public MiTareaPeriodica
      (time plazo, time tiempoEjec, priority prio, time period, int id)
   {
      super(tiempoEjec, prio);
      this.relDeadline=plazo;
      this.period=period;
      this.id=id;
   }

   // código a ejecutar en cada instancia
   public void run() {
      System.out.println("Tarea "+id+". Instancia: "+contador);
      contador++;
      // para probar el funcionamiento de la detección de incumplimiento de plazos,
      // cada 10 veces vamos a tardar más
      if (contador%10==0) {
```

```
            System.out.println("Durmiendo 10 segundos en Tarea "+id);
            Thread.sleep(3000); // tiempo en ms
            System.out.println("Despertando Tarea "+id);
        } catch (InterruptedException e) {
             // nada que hacer
        }
    }

    // Código a ejecutar si se incumple el plazo
    public void deadlineMiss() {
        System.out.println("Plazo incuplido!!!!!!!!!! En tarea "+id
                           + "instancia: "+contador);

    }

}

// crear jobs
MiTareaPeriodica t1=new MiTareaPeriodica(1.0, 0.1, 10, 1.0, 1);
MiTareaPeriodica t2=new MiTareaPeriodica(2.5, 0.1, 8, 2.5, 2);

// crear mecanismos de activación
PediodicRelease r1=new PeriodicRelease(t1);
PediodicRelease r2=new PeriodicRelease(t2);

// crear executors
Executor e1=new Executor(r1,t1,20);
Executor e2=new ExecutorWithDeadlineTermination(r2,t2,20);

// arrancar las tareas
e1.run();
e2.run();
```

IV.1.10. Usos Conocidos

Tareas repetitivas que deben ejecutarse de manera periódica, y con detección de incumplimiento de plazos.

IV.1.11. Patrones relacionados

AperiodicTask, SporadicTask

IV.2 Patrón **SporadicTask:**

IV.2.1 Propósito

Ejecuta repetidamente un trabajo en respuesta a la llegada de un evento externo, cuyo intervalo mínimo entre llegadas está especificado. El trabajo puede tener un plazo de finalización que se inicia con la llegada del evento y termina con la finalización de la ejecución del trabajo. El patrón proporciona mecanismos para

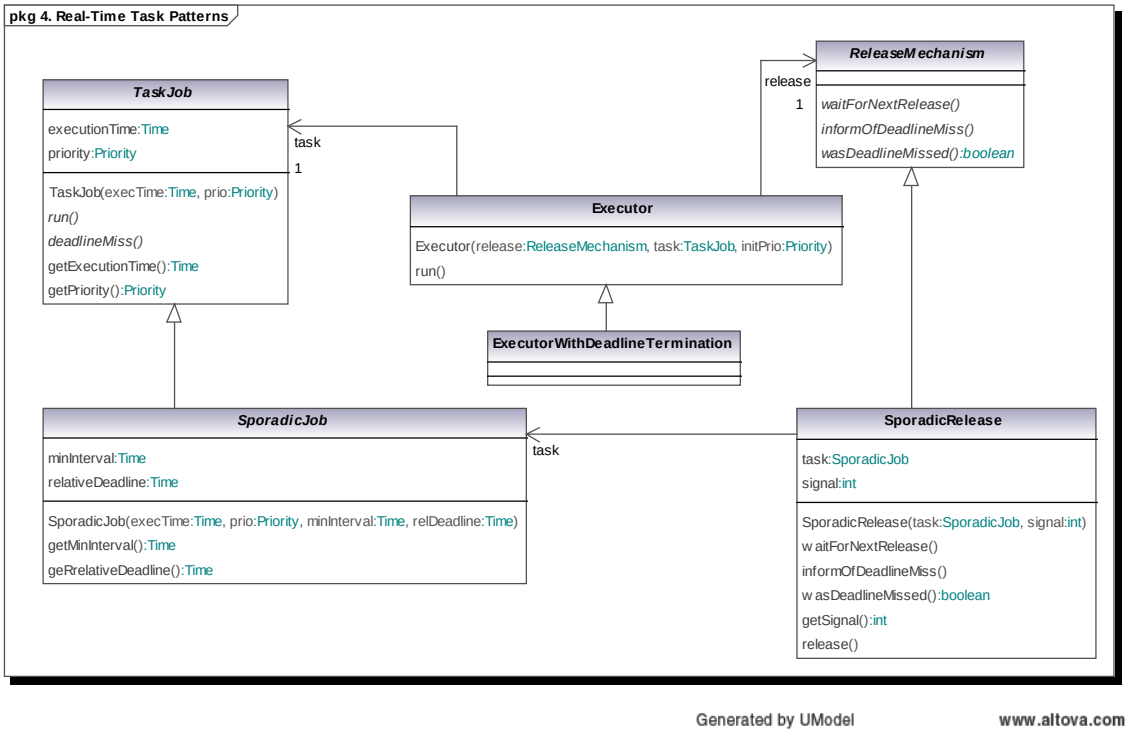
detectar incumplimientos de este plazo. Asimismo proporciona los mecanismos para proteger al sistema de llegadas de eventos demasiado próximas, garantizando así la ejecución con una separación suficiente y, por tanto, con un gasto máximo de CPU igual al de procesar un evento a cada intervalo igual al tiempo mínimo entre llegadas.

IV.2.2 Motivación. Prácticamente todas las aplicaciones de tiempo real tienen operaciones que deben responder a eventos externos cuya temorización se desconoce, pero en los que está establecido por medio de mecanismos hardware o software un intervalo mínimo entre activaciones. Es habitual que estas tareas tengan un plazo de finalización para cada una de sus activaciones, que se cuenta desde la llegada del evento.

IV.2.3 Aplicabilidad. En subsistemas de control o de monitorización de señales.

IV.2.4 Estructura.

La figura muestra un diagrama de clases de los elementos involucrados.



IV.2.5 Elementos

e) SporadicJob: Es una especialización del TaskJob que contiene los parámetros que caracterizan los requisitos temporales (mínimo intervalo entre activaciones, plazo relativo y tiempo de ejecución de peor caso), los requisitos de planificación (prioridad), el código a ejecutar en cada instancia (run()) y el código a ejecutar si se detecta un incumplimiento del plazo (deadlineMiss())

El SporadicJob es una clase abstracta que requiere ser extendida. En la extensión deben incluirse atributos que contengan el estado de la tarea que debe preservarse de una instancia a la próxima. El constructor puede redefinirse para inicializar convenientemente este estado. Deben redefinirse los métodos abstractos run() y deadlineMiss().

f) SporadicRelease: Es una especialización del ReleaseMechanism que recibe activaciones externas a través de su método público release(). Su constructor requiere que se le pase un SporadicJob y la señal a utilizar para comunicar el incumplimiento de plazo (o el mecanismo de notificación apropiado, si no se desea utilizar una señal).

   El SporadicRelease es una clase concreta que se suministra como parte de la implementación del patrón. Sus operaciones (con la excepción de release()) son para uso exclusivo de la implementación, en particular del Executor asociado.

g) Executor: Proporciona el thread que ejecuta la tarea esporádica. A su constructor hay que pasarle el SporadicJob y el SporadicRelease a utilizar.

   El Executor es una clase concreta que se suministra como parte de la implementación del patrón.

h) ExecutorWithDeadlineTermination. Es una extensión del Executor que, al igual que éste, proporciona el thread que ejecuta la tarea periódica. La principal diferencia es que establece los mecanismos necesarios para abortar la ejecución de la instancia actual del TaskJob si se detecta que ha incumplido su plazo.

   El ExecutorWithDeadlineTermination es una clase concreta que se suministra como parte de la implementación del patrón.

IV.1.6 Colaboraciones

La operación run() del Executor invoca constantemente a waitForNextRelease() del SporadicRelease, para esperar al próximo periodo, y seguidamente al run() del SporadicJob(), para ejecutar una instancia de la tarea periódica.

IV.2.7 Consecuencias

Este patrón constituye la manera más básica de ejecutar una tarea repetitiva que ejecuta una instancia de forma esporádica, incluyendo la posibilidad de detectar el incumplimiento del plazo de cada instancia, así como un mecanismo para asegurar la separación mínima entre activaciones.

IV.2.8 Implementaciones

Executor en POSIX: ver sección IV.1.8

ExecutorWithDeadlineTermination En POSIX: ver sección IV.1.8

SporadicRelease en POSIX

```
// atributos adicionales
boolean deadlineWasMissed
time minInterval
time lastTime ; // hora de llegada del anterior evento
time deadline;
pthread_mutex_t m;
pthread_cond_t cv;
cola-de-valores-de-tiempo llegadas;


SporadicRelease(PeriodicJob task, int signal) {
   this.task=task;
   this.signal=signal;
   this.minInterval=task.getMinInterval();
   this.deadline=task.getRelativedeadline();
   this.deadlineWasMissed=false;
   initialize timer based on CLOCK_MONOTONIC, with an action set to send
      this.signal;
   inicializar el mutex m con protocolo PRIO_PROTECT y
       techo de prioridad ceiling
   inicializar la variable condicional cv
   llegadas.initialize();
}

void release() {
   get_time(CLOCK_MONOTONIC,ABS_TIME,now);
   m.lock()
   // insertar hora de llegada en la cola de llegadas
   llegadas.insert(now);
   cv.signal();
   m.unlock();
}

void waitForNextRelease() {
   timer_disarm();
   get_time(CLOCK_MONOTONIC, now);
   deadlineWasMissed=now-lastTime>relativeDeadline;

   // wait for next event
   m.lock();
   while (llegadas.estaVacia())
      cv.wait(m);
   }
   time currentTime=llegadas.Extrae()
   m.unlock();

   // check and enforce minimum separation
   clock_gettime(CLOCK_MONOTONIC,ABS_TIME,now);
   if (now-lastTime<minInterval) {
      //not enough separation
      clock_nanosleep(CLOCK_MONOTONIC,ABS_TIME,lastTime+minInterval);
   }
   lastTime=currentTime;
}

boolean wasDeadlineMissed() {
    return deadlineWasMissed;
}
```

```
void informOfDeadlineMiss() {
   timer_arm(timer,lastTime+deadline);
}

int getSignal() {
   return this.signal;
}
```

IV.2.9 Forma de uso

A continuación se muestran los pasos necesarios para utilizar este patrón.

f) Extender la clase SporadicJob:
   a. Añadir los atributos que contendrán el estado de la tarea periódica que debe preservarse de una instancia a la siguiente.
   b. Se puede redefinir el constructor añadiéndole los parámetros que se necesiten para inicializar el estado de la tarea esporádica. El constructor debe invocar al constructor del SporadicJob para inicializar los parámetros básicos, y luego ejecutar las instrucciones de inicialización que se deseen.
   c. Redefinir la operación run() que es el código que se ejecutará a cada instancia
   d. Redefinir la operación deadlineMiss(), que será invocada si hay un incumplimiento de plazo

g) Crear un objeto de la nueva clase extensión de SporadicJob. Lo llamaremos tarea. Al crearlo, pasarle los parámetros temporales (prioridad, plazo, tiempo de ejecución, tiempo mínimo entre llegadas), los parámetros de planificación (prioridad) y cualquier otro definido en la extensión.
h) Crear un objeto de la clase PeriodicRelease, pasándole como parámetro al constructor el objeto tarea y el número de señal a utilizar para la notificación del incumplimiento de plazo. Llamaremos al nuevo objeto release.
i) Crear un objeto de la clase Executor o ExecutorWithDeadlineTermination (según el comportamiento deseado), pasándole como parámetros los objetos tarea y release.
j) Invocar el método run() del Executor para arrancar la terea periódica, que funcionará a partir de ahora de forma indefinida.

IV.2.10 Ejemplo

En este ejemplo describimos una aplicación Java con dos tareas esporádicas que, a modo de ejemplo, comparten el mismo código y ponen mensajes personalizados en pantalla de forma periódica.

```
// Extensión de SporadicJob que contiene el estado de la tarea periódica
// y el código a ejecutar en cada instancia

public class MiTareaEsporadica extends PeriodicJob {

   // atributo que cuenta el número de instancias ejecutadas
   private int contador=0;
```

```
    // atributo que contiene el identificador de la tarea
    private int id;

    // constructor al que se le pasan el plazo, tiempo de ejecución, prioridad e
    // intervalo mínimo del Job

    public MiTareaEsporadica
        (time tiempoEjec, priority prio, time minInterval, time plazo, int id)
    {
        super(tiempoEjec, prio);
        this.relDeadline=plazo;
        this.minInterval=minInterval;
        this.id=id;
    }

    // código a ejecutar en cada instancia
    public void run() {
        System.out.println("Tarea "+id+". Instancia: "+contador);
        contador++;
        // para probar el funcionamiento de la detección de incumplimiento de plazos,
        // cada 10 veces vamos a tardar más
        if (contador%10==0) {
            System.out.println("Durmiendo 10 segundos en Tarea "+id);
            Thread.sleep(3000); // tiempo en ms
            System.out.println("Despertando Tarea "+id);
        } catch (InterruptedException e) {
             // nada que hacer
        }
    }

    // Código a ejecutar si se incumple el plazo
    public void deadlineMiss() {
         System.out.println("Plazo incuplido!!!!!!!!!! En tarea "+id
                                + "instancia: "+contador);

    }

}

// crear jobs
MiTareaEsporadica t1=new MiTareaEsporadica(1.0, 0.1, 10, 1.0, 1);
MiTareaEsporadica t2=new MiTareaEsporadica(2.5, 0.1, 8, 2.5, 2);

// crear mecanismos de activación
SporadicRelease r1=new PeriodicRelease(t1);
SporadicRelease r2=new PeriodicRelease(t2);

// crear executors
Executor e1=new Executor(r1,t1,20);
Executor e2=new ExecutorWithDeadlineTermination(r2,t2,20);

// arrancar las tareas
e1.run();
e2.run();

// ahora habría que ir llamando sucesivamente a r1,release() y r2.release()
// para ir generando los eventos.
```

IV.2.11. Usos Conocidos

Tareas repetitivas que deben ejecutarse de manera periódica, y con detección de incumplimiento de plazos.

IV.2.12. Patrones relacionados

AperiodicTask, PeriodicTask

IV.3 Patrón **AperiodicTask**

IV.3.1 Propósito

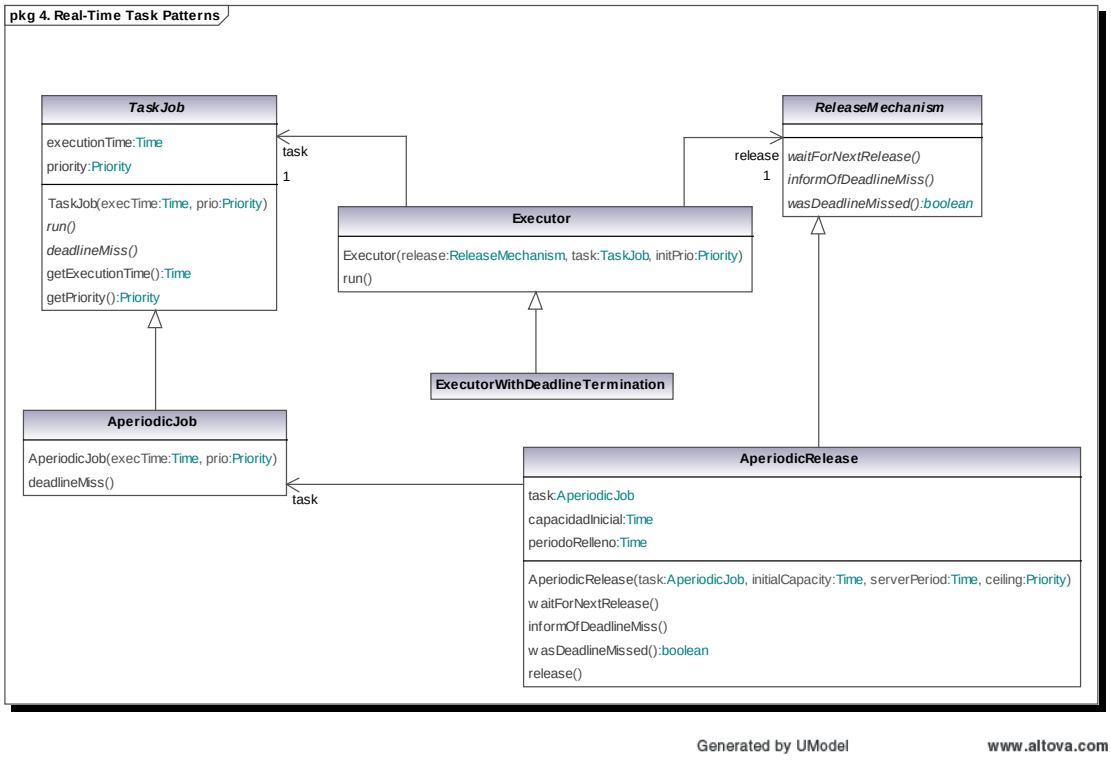
Ejecuta un trabajo que se solicita a intervalos irregulares y sin requisitos temporales. Las tareas son ejecutadas bajo un servidor de tiempo de ejecución limitado, de modo que su impacto sobre tareas de prioridad inferior se pueda limitar.

IV.3.2 Motivación. Prácticamente todas las aplicaciones de tiempo real tienen necesidad de responder a estímulos repetitivos que llegan del exterior, quizás a través de una interrupción hardware o de un mensaje que llega por una red, siendo los intervalos entre las llegadas de estos estímulos irregulares. Estas tareas no pueden tener un plazo de finalización para sí mismas, dada la irregularidad de sus ritmos de activación, pero es necesario limitar la cantidad de recursos o anchura de banda que consumen, para garantizar que otras tareas de prioridad inferior tienen suficientes recursos para ejecutarse cumpliendo sus requisitos temporales.

IV.3.3 Aplicabilidad. En subsistemas de control o de monitorización de señales en los que hay respuestas a eventos externos, por ejemplo provenientes de un operador o de sucesos de temporización irregular.

IV.3.4 Estructura.

La figura muestra un diagrama de clases de los elementos involucrados.



IV.3.5 Elementos

i) AperiodicJob: Es una especialización del TaskJob que contiene los parámetros que caracterizan los requisitos temporales (tiempo de ejecución de peor caso), los requisitos de planificación (prioridad), el código a ejecutar en cada instancia (run()) y el código a ejecutar si se detecta un incumplimiento del plazo (deadlineMiss()) que en este caso es nulo, ya que esta funcionalidad no es necesaria en tareas aperiódicas.

El AperiodicJob es una clase abstracta que requiere ser extendida. En la extensión deben incluirse atributos que contengan el estado de la tarea que debe preservarse de una instancia a la próxima. El constructor puede redefinirse para inicializar convenientemente este estado. Debe redefinirse el método abstracto run().

j) AperiodicRelease: Es una especialización del ReleaseMechanism que ofrece un mecanismo de sincronización para recoger los eventos aperiódicos y activar la tarea aperiódica bajo el control de un servidor que limita la anchura de banda. En este caso se ha optado por el servidor más sencillo, que es el de muestreo periódico. Su constructor requiere que se le pase un AperiodicJob, así como los parámetros del servidor: capacidad inicial de ejecución (que debe ser mayor o igual que el tiempo de ejecución de peor caso de la tarea) y periodo del servidor de muestreo periódico.

El AperiodicRelease es una clase concreta que se suministra como parte de la implementación del patrón. Su operación release() es utilizada por otras tareas del sistema, incluso por una rutina de interrupción, para avisar de la llegada de

un evento aperiódico. El resto de sus operaciones son para uso exclusivo de la implementación, en particular del Executor asociado.

k) Executor: Proporciona el thread que ejecuta la tarea aperiódica. A su constructor hay que pasarle el AperiodicJob y el AperiodicRelease a utilizar.

   El Executor es una clase concreta que se suministra como parte de la implementación del patrón.

l) ExecutorWithDeadlineTermination. Es una extensión del Executor que, al igual que éste, proporciona el thread que ejecuta la tarea aperiódica. Para tareas aperiódicas no hay diferencia de comportamiento con respecto al Executor.

   El ExecutorWithDeadlineTermination es una clase concreta que se suministra como parte de la implementación del patrón.

IV.3.6 Colaboraciones

La operación run() del Executor invoca constantemente a waitForNextRelease() del PeriodicRelease, para esperar al próximo periodo, y seguidamente al run() del PeriodicJob(), para ejecutar una instancia de la tarea periódica.

IV.3.7 Consecuencias

Este patrón constituye la manera más básica de ejecutar una tarea aperiódica que ejecuta una instancia cada vez que llega un evento del exterior, pero limitando la anchura de banda dedicada a la tarea aperiódica, de modo que otras tareas de prioridad inferior puedan ejecutar cumpliendo sus requisitos temporales.

IV.3.8 Implementaciones

Executor en POSIX: ver sección IV.1.8

ExecutorWithDeadlineTermination En POSIX: ver sección IV.1.8

AperiodicRelease en POSIX

```
// atributos internos
int maxActivations;
int pendingActivations; // activaciones por procesar en este periodo
int  pendingEvents;       // eventos pendientes de procesar
boolean primeraVez, wasBlocked;
timespec nextPeriod ; // hora para el próximo muestreo

AperiodicRelease(AperiodicJob task, Time initialCapacity,
                    Time serverPeriod, Priority ceiling)
{
   this.task=task;
   this.initialCapacity=initialCapacity;
   this.serverPeriod=serverPeriod;
   this.primeraVez=true;
```

```
    this.wasBlocked=true;
    this.maxActivations=suelo(initialCapacity/task.getExecutionTime());
    this.pendingActivations=0;
    this.pendingEvents=0;
}

void waitForNextRelease() {
     if (primeraVez) {
          //anotar el principio del periodo
          this.nextPeriod=get_time(CLOCK_MONOTONIC);
          primeraVez=false;
     }
     if (wasBlocked) {
           pendingActivations=min(pendingEvents,maxActivations)
           wasBlocked=false;
     }
     If (pendingActivations>0) {
        // procesar una activación
        pendingActivations--;
     } else {
        // dormir hasta el siguiente periodo
        wasBlocked=true;
       nextPeriod=nextPeriod+serverPeriod;
       clock_nanosleep(CLOCK_MONOTONIC,ABS_TIME, nextPeriod);
       //comenzar a procesar eventos en el nuevo periodo
       this.waitForNextRelease();
     }
}

void release() {
     // no necesita protección con mutex si
     // se puede implementar con una variable atómica
     pendingEvents++;
}

boolean wasDeadlineMissed() {
   // en esta mplementación no se comprueban los deadlines
  return false;
}

void informOfDeadlineMiss() {
   // no hay que hacer nada, pues no se comprueban los deadlines
}
```

AperiodicJob: basta proporcionar una implementación nula para deadlineMiss(), ya que en este caso no se usa

IV.3.9 Forma de uso

A continuación se muestran los pasos necesarios para utilizar este patrón.

k) Extender la clase AperiodicJob:
   a. Añadir los atributos que contendrán el estado de la tarea aperiódica que debe preservarse de una instancia a la siguiente.
   b. Se puede redefinir el constructor añadiéndole los parámetros que se necesiten para inicializar el estado de la tarea periódica. El constructor

debe invocar al constructor del AperiodicJob para inicializar los parámetros básicos, y luego ejecutar las instrucciones de inicialización que se deseen.

c. Redefinir la operación run() que es el código que se ejecutará a cada instancia

l) Crear un objeto de la nueva clase extensión de AperiodicJob. Lo llamaremos tarea. Al crearlo, pasarle los parámetros temporales (tiempo de ejecución), los parámetros de planificación (prioridad) y cualquier otro definido en la extensión.

m) Crear un objeto de la clase AperiodicRelease, pasándole como parámetro al constructor el objeto tarea y los parámetros del servidor (capacidad inicial y periodo del servidor). Llamaremos al nuevo objeto release.

n) Crear un objeto de la clase Executor (el ExecutorWithDeadlineTermination no tiene sentido en este caso, pues no se comprueban los deadlines), pasándole como parámetros los objetos tarea y release.

o) Invocar el método run() del Executor para arrancar la terea periódica, que funcionará a partir de ahora de forma indefinida.

IV.3.10 Ejemplo

En este ejemplo describimos una aplicación Java con dos tareas aperiódicas que, a modo de ejemplo, comparten el mismo código y ponen mensajes personalizados en pantalla de forma periódica.

```
// Extensión de AperiodicJob que contiene el estado de la tarea aperiódica
// y el código a ejecutar en cada instancia

public class MiTareaAperiodica extends AperiodicJob {

   // atributo que cuenta el número de instancias ejecutadas
   private int contador=0;
   // atributo que contiene el identificador de la tarea
   private int id;

   // constructor al que se le pasan el tiempo de ejecución, prioridad, e
   // identificador del Job

   public MiTareaAperiodica
      (time tiempoEjec, priority prio, int id)
   {
      super(tiempoEjec, prio);
      this.id=id;
   }

   // código a ejecutar en cada instancia
   public void run() {
      System.out.println("Tarea "+id+". Instancia: "+contador);
      contador++;
   }

}

// crear jobs
MiTareaAperiodica t1=new MiTareaAperiodica(0.1, 10, 1);
MiTareaAperiodica t2=new MiTareaAperiodica(0.15, 8, 2);
```

```
// crear mecanismos de activación
ApediodicRelease r1=new AperiodicRelease(t1,0.3,1.5);
ApediodicRelease r2=new AperiodicRelease(t2, 0.5, 2.5);

// crear executors
Executor e1=new Executor(r1,t1,20);
Executor e2=new Executor(r2,t2,20);

// arrancar las tareas
e1.run();
e2.run();
```

IV.3.10. Usos Conocidos

Tareas repetitivas que deben ejecutarse de manera aperiódica, y con limitación del consumo de CPU por intervalo temporal configurable.

IV.3.11. Patrones relacionados

PeriodicTask, SporadicTask